Pro IronPython

Alan Harris

Apress®

Pro IronPython

ISBN-13 (pbk): 978-1-4302-1962-0

ISBN-13 (electronic): 978-1-4302-1963-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Mark Beckner, Jonathan Hassel
Technical Reviewer: Shawna Garver
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editor: Elliot Simon
Associate Production Director: Kari Brooks-Copony
Production Editor: April Eddy
Compositor: Linda Weidemann, Wolf Creek Publishing Services
Proofreaders: Linda Seifert and Kim Burton
Indexer: Julie Grady
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

Contents at a Glance

Contents

About the Author



ALAN HARRIS is a developer at the Council of Better Business Bureaus in Arlington, Virginia, where he works feverishly on the content management systems and search engine optimization initiatives. He has been working with the .NET framework since 2002 and has an admitted preference for C#, though he wishes those VB .NET folks could get a little more respect sent their way. In a previous life he worked for a naval subcontractor, writing firmware in C to allow custom safety hardware to communicate via the ORBCOMM satellite network; at a nonprofit, migrating legacy code to .NET; and on cost-analysis tools for industry use. He keeps his F# experience tucked away as a secret weapon.

When not parked in front of a computer of some kind, he is an avid practitioner of Krav Maga, has been a drummer and percussionist for more than 20 years, can't seem to stay out of Gold's Gym, and has a demonstrated capability to watch Akira Kurosawa's classic *Seven Samurai* more than once in a single day. He also has a longstanding bet on the true nature of the Smoke Monster in *Lost*.

About the Technical Reviewer



SHAWNA GARVER started working as a professional FORTRAN programmer for SAIC at the age of 16 while attending the University of Maryland. On her first day, she tried to use her computer, only to discover it would not start. Using her advanced problem-solving skills, she began to push the power button on and off like mad. It was then she discovered the latest technological advance in 1980s-era computing—the power strip! Voila, problem solved! Shawna went on to win an all-tuition-and-fees scholarship to one of the nation's top engineering programs and to graduate with dual degrees in engineering and architecture. Since then she has worked as an engineer and a project manager in the maritime and aerospace industries, with specialty in programming, data analysis, database design, and web applications. Shawna lives near Washington, D.C., with her husband, Chris, and her two children.

Acknowledgments

My family and friends are my lifeline to the outside world, which apparently contains a mysterious glowing orb that occasionally seeks to be my undoing. I thank them from the bottom of my heart for their support and encouragement as well as for countless hours of sheer entertainment. I'll win that war against the giant glowing pain orb eventually.

To Mark Beckner, Beth Christmas, Jonathan Hassell, Elliot Simon, and Shawna and Chris Garver, thank you so much for putting up with my ability to turn something in precisely one day later than you asked. More specifically, thank you for the time and effort you put in to helping me write something that fills a void on my shelf and hopefully the shelves of many other developers wondering what exactly this IronPython business is all about.

I couldn't think of a better way to relieve a little stress than spending some time with the folks at Krav Works in Falls Church, Virginia. Vince, you're an excellent instructor; I now keep my hands in front of my face at inappropriate times, just in case a punch comes out of nowhere.

Finally, George the Cat: get off my countertops. Seriously: you're the best cat ever, but kitties are not for countertops. I looked it up.

Introduction

I come from a background of static typing and rigid languages: C, C++, C#. I'm seeing more than one trend at work here. For the longest time I felt something of a warm, fuzzy sensation when it came to my programming. The data type on the left matched the data type on the right. All was well. Then Python walked through the door.

Python's not the only game in town to use dynamic typing (also known as *duck typing*, which you will learn about in due time), but it did catch my eye and challenged my perspective as a programmer. "What is this? How does one accurately program anything in this fashion? Five hundred lines of code and not one duck! Python's a liar." A little unsettling, you can imagine.

It's taken me some time to get really comfortable with the notions behind Python and, by extension, IronPython. The effort was not wasted. IronPython is powerful, fast, and a first-rate language supported fully by Microsoft, enabling developers to get their work done better, faster, and cleaner. In the end, a good measurement of a programming language is how elegantly you can express your intentions in code while still achieving the functionality you desired. I think you'll be pleasantly surprised with IronPython.

I'm not asserting that you're going to be an instant convert to the ways of the Pythonistas. I'm simply asking you to try. If you're coming from a programming background, particularly one with more rigid rules, jump into the deep end of the pool for a bit. Try something scary. I think you'll find that type errors really don't crop up too often, I might argue that it requires a somewhat more careful developer. But the freedom of flexibility is perhaps not even as significant a benefit as is some added attention on your part. The net result should be better code all around; happier developers, happier users. If you have no programming background whatsoever, come on in anyway. I love a blank slate.

A moment ago I mentioned *duck typing*, a concept that comes up a lot in IronPython: if it looks like a duck and quacks like a duck, it must be a duck. This fundamental idea, when applied to data types and objects, allows a significant amount of polymorphism to be baked right into the language itself from the outset. Many developers take issue with this and find the approach too loose, too error-prone. I was speaking to Pythonista and author Michael Foord about the matter. I mentioned the argument "What if it looks like a duck, quacks like a duck, but is really a dragon impersonating a duck? You don't want a dragon in your pond." He replied, "If you code dragons, you've got no one to blame but yourself if they get in your pond." I informed him that "the only difference between my IronPython dragons and my C# ones is that my C# dragons have 'Hello, my name is' nametags." So it goes.

Is This for Me?

IronPython and the .NET framework are very approachable to new developers. The tools are free, and there is an overabundance of both documentation and skilled developers who are happily sharing their knowledge with the world. The barrier to entry is supremely low these days.

If this is your first programming book, so be it! Come along for the ride. You'll see both sides of the programming fence, for you'll find examples here in IronPython and C# as well as an entire chapter devoted to getting the two to play happily and nicely with one another. I also cover many basic programming fundamentals as well as the advanced stuff. You'll get exposed to multiple languages and the .NET framework by the time we're through.

If you're already versed in Python but not in .NET, you might just find that you can get your programming tasks done a lot more easily with the tested and powerful .NET framework behind you.

If you already know both IronPython and .NET, this book should make for a good reference of various tricks and techniques, particularly in the realms of language integration and web development.

An Overview of This Book

Being an IronPython developer can mean a lot of things. You could write software to be run via the command line, as a Windows Forms application, or as a web application. That means we have a lot of ground to cover. We need both to address IronPython syntax as well as to look at how it fits into the larger .NET framework.

Chapter 1: Introduction to IronPython

The introductory chapter provides you with a little background on Python and IronPython as well as on the .NET framework itself. We'll look at what constitutes a dynamic language and contrast it with a static one. Then we'll get ourselves a copy of IronPython and immediately try our hand at a sample and see how the language works.

Chapter 2: IronPython Syntax

IronPython has a rich but straightforward syntax and many built-in functions that make your life easier as a developer and ensure you don't have to reinvent the wheel. This chapter looks at that syntax but does not yet cover interaction with the larger .NET framework. In fact, it will become apparent that you can actually write entire IronPython applications that don't really make use of the framework at all, allowing Python developers to ease into the .NET world quite easily and gradually.

Chapter 3: Advanced IronPython

As with most programming languages, you can use the simplest syntax to express the most complicated ideas. In this chapter we'll expand what we know and look at more complex data constructs, base classes, and object-oriented design principles.

Chapter 4: IronPython Studio

This chapter focuses on IronPython Studio and how you can use it to speed your development process. Up until this chapter the code has been entered entirely using the command-line IronPython interpreter. It's time to kick things up a notch and begin working with the Integrated Development Environment and also to begin working with Windows Forms applications.

Chapter 5: Mixing and Mingling with the CLR

It's difficult to really know and understand a language until you've built something with it, hit some walls, and learned how to take an application from design to implementation. In this chapter we'll begin making heavy use of the .NET framework and build the distant cousin of a very familiar application from the ground up. We'll pay special attention to points where the .NET framework can save us time and energy, especially when coupled with the IronPython Studio IDE.

Chapter 6: Advanced Development

One of the coolest things about IronPython is how easily it can be used with other .NET languages. This chapter is all about how to employ IronPython as a scripted plug-in manager in a C# application. The plug-in system is designed to be straightforward and simple, and it should prove to be a good starting point for your own improvements and customizations. It can really save you endless hours of work if the need arises for extensibility in an existing application (or if you just want to add something neat like that at the very beginning).

Chapter 7: Data Manipulation

This chapter covers communicating with SQL Server and how to use Structure Query Language (SQL) to work with the database via IronPython code. I've also provided advice on how to protect yourself and your users against malicious entities who might try to use specially crafted SQL to circumvent your security.

Chapter 8: Caught in a Web

If you're interested in web development, search engine optimization, and standards compliance, this chapter will be of special interest because it provides insight into all these areas and how IronPython helps you achieve the results you want. You'll find useful tips like how to do cross-page PostBacks, how to prevent arbitrary code injection, and more.

Chapter 9: IronPython Recipes

This final chapter provides a lot of varied snippets for many aspects of console, desktop, and web development, ranging from design patterns to search engine optimization tips, along with a final message for readers who kindly explored IronPython with me.

Obtaining This Book's Source Code

While I'm a believer in hands-on learning and dutifully type every line of code in the books that I read, I know there's at least one person out there thinking, "I do not type nearly fast enough even to consider that a possibility." No worries: the code examples in this book are available as a free download from the Source Code/Download area of the Apress website at `http://www.apress.com`. Look up this book by its name, *Pro IronPython*, to find the appropriate downloads.

Obtaining Updates for This Book

My being blessed with terrible vision results in a simple truth: despite having four eyes, I've likely missed something along the way. The wonderful editors do their best to keep up with my erratic keyboard pounding, but even they are only human, and the burden of guilt lies squarely with me. The Apress website maintains a list of errata and provides a way to notify me of errors that might pop up after you have this book in your hands.

Contacting Me

I'm an outgoing type of guy, and I love to talk to other developers. I currently maintain status updates of the minutiae of my life at Twitter; feel free to follow me at `http://twitter.com/Anachronistic`. Alternatively, if 140 characters just isn't enough to get those thoughts out, you are completely welcome to contact me at `dotnetalan@gmail.com`. I'm a web developer by day, Kravist by night; if I don't get back to you immediately, I promise I'm not deliberately blowing you off. You can always drop me a quick reminder that you need a little attention, too.

CHAPTER 1

Introduction to IronPython

"Snakes. Why'd it have to be snakes?" — Indiana Jones

This is a great time to be a .NET developer. Software architects and engineers have a fantastic toolkit at their disposal that allows them to produce quality code quickly. This book is about IronPython and how you can fit it into your toolkit to solve the issues you face as a developer.

IronPython represents a very new offering from Microsoft that works alongside the other .NET family of languages, adding the power and flexibility that comes with a dynamic language such as Python. To understand where IronPython fits into the scheme of things, let's go back to the origins of Python and see how we got to the present day. If you're coming from a software development background from another language, such as C# or VB.NET, this history should help clarify some of the design decisions about the Python language, which, although very different from many other languages in use today, results in a powerful, flexible, and rapid development tool.

A Humble Beginning

Python's origins date back to the 1980s. A developer named Guido van Rossum created Python to be the successor to a language called ABC. The idea was that the Python language would be extremely readable and not cluttered with confusing syntax and markup. Blocks of code are denoted by whitespace indentation, variables are strongly typed, and it would not try to force developers to learn and implement any one particular programming style. For example, Python developers had at their disposal the language features necessary to move between functional, object-oriented, and structured programming, and more. The language is quite capable of adapting to the individual developer's needs with an expanding array of add-ins and an active user community.

Note Guido has remained very active in the Python community over the years and has since had bestowed on him the lofty mantle of "Benevolent Dictator for Life," a title that is indeed well earned.

Over the years, Python has proven itself to be quite a capable language, powering a wide variety of high-visibility web sites, including YouTube and Google. It has a reputation for being easy to work with and for allowing applications to be highly available without requiring a large team of developers to create and maintain them. As such it represents an attractive language choice for companies looking to create an online presence quickly or for those looking to improve their existing back-end infrastructure with the benefits of Python.

Jython: A Taste for Java

Although Python by itself is a powerful language, there have been implementations of it in the past that aim to make use of other languages within Python, thereby blending the best of all of these. The most notable predecessor to IronPython is Jython, created in 1997 by Jim Hugunin (who would eventually go on to create IronPython, but we're jumping ahead here!). Jython's strength lies in its ability to call and use Java classes natively, thereby expanding the Python language. This is a very important point in the IronPython story, and I'll emphasize it again: *Jython can call and use Java classes natively.* We'll examine this point again shortly.

IronPython: "Import .NET"

In 2004, the Microsoft .NET framework and, in particular, the Common Language Runtime (CLR) were really starting to make waves in the software development world. The .NET framework and CLR present developers with a way to write code in their language of choice, so long as there exists a compiler that can translate the source code into Common Intermediate Language (CIL) bytecode. Developers can write for the .NET framework in any language they like. It is in this .NET-and-CLR platform environment that IronPython was created.

After leaving the Jython project, the .NET framework and CLR caught Jim Hugunin's eye. He began working with the CLR with the intention of creating an article titled "Why .NET Is a Terrible Platform for Dynamic Languages." To his surprise, the framework turned out to perform very well, and his focus shifted. He decided to create IronPython, a .NET equivalent of Jython. In 2004, he joined the Microsoft CLR team to work on IronPython full time, with the support and resources of the software giant behind him.

Let's not mince words here: IronPython is a first-class language, supported by Microsoft and a growing user community, built on the backs of the hardworking (and very experienced) Python community and fueled by their continual input and refinement. Releases are frequent and stable and further the capabilities of the language and what developers can accomplish with it. Python is by no means a new language;

IronPython gains the benefit of those many years of development experience and successful projects, which puts it quite far ahead in any programming language race.

Why Is .NET Important?

The mixing and mingling of Python and the .NET framework is powerful. The .NET framework is a significant offering from Microsoft; it is the platform of choice for many developers when it comes to building desktop and web software. One need look no further than language independence as a selling point. Gone are the days when a developer finds that he or she needs to learn a multitude of language nuances to be productive or solve a task. A studied Visual Basic .NET developer can easily pick up C#, F#, or IronPython; because the .NET framework unifies these languages with a common infrastructure, language choice is down to preference and comfort, not necessity. This design architecture is what allows something like IronPython to exist in the first place. Each language has strengths and weaknesses, and certain languages *do* perform some tasks easier or in a more straightforward manner. We will examine some of the particular strengths of IronPython over other .NET languages throughout the book, and I will provide the occasional C# sample for comparison.

Caution It's worth mentioning at this point that IronPython is designed to implement CPython 2.5.2 for version compatibility, but not everything written in IronPython will work in CPython, and vice versa. There are a few underlying language differences, which we will cover throughout the book, and I will flag known issues when we encounter them. As we progress, just keep in the back of your mind the notion that IronPython is its own separate language.

What Exactly Is IronPython?

IronPython is a dynamic language, an implementation of the Python language that is written in C# and built to run on the Microsoft .NET 2.0 (or greater) framework. We'll cover dynamic versus static languages in a bit. By investing time and energy into learning how to write code in IronPython, you're expanding your skill set by a greater amount than you might realize at a first glance. The Common Language Runtime is the underlying technology of .NET, combined with the Common Intermediate Language and the Common Language Infrastructure. Since IronPython is implemented in .NET 2.0, you are taking advantage of and learning to use this underlying set of technologies as you go. The IronPython way of creating a Windows form programmatically happens to be very similar

to the way it would look and be done in C# or Visual Basic .NET. The reason for this is the underlying .NET framework these languages use; you're getting more bang for your buck! What you learn by teaching yourself IronPython gives you a leg up in learning other .NET languages if you choose to do so, and it's hard to argue that learning more in less time is a poor decision!

What Can IronPython Do for Me Today?

Unless you're looking at IronPython from a strict hobbyist perspective, I would be willing to wager that one of the biggest questions you have is what IronPython can do to make your programming time more productive? The short answer is "a lot." It sounds cliché, but there really is a lot under the hood in terms of programming power and elegant code. In fact, the Python language is considered so readable that many people refer to it as "executable pseudocode."

■ **Note** *Pseudocode* is a fancy term for code intended for human, not machine, use, and it generally looks something like the language in which the final code will be written, but without the messy details. A good analogy for pseudocode is scribbling out a drawing on a napkin: it doesn't have to look perfect; rather, it simply needs to convey the intended design to someone else. When people refer to Python code as executable pseudocode, it's really a statement about how readable the Python language is. It lacks so much of the markup that many languages have that it looks like pseudocode, but it is in fact executable code.

- *Rapid prototyping*: IronPython allows developers to design and test ideas quickly, either in scripts or using the interactive interpreter. No massive compilation times are required!
- *Easy integration*: It's extremely easy to integrate IronPython code in other applications, both of the "commercial, off-the-shelf" variety and the "I made it myself" variety. Many commercial products use Python as a scripting language; by learning IronPython, you can build and customize aspects of those programs as well. We will be using IronPython to customize other .NET applications later in the book.

- *Extensible*: The Python community is vibrant, thriving, and large. It's extremely easy to add new functionality to your IronPython applications with community-driven code. Flexibility is key; IronPython is designed to make your programming life easier!
- *Style convenience*: IronPython does not require you to be an object-oriented programmer or a functional programmer or any other type you can think of. IronPython allows a variety of programming constructs. If you're more comfortable with F# than C#, IronPython will happily allow you to program in a functional versus object-oriented manner.

Yes, But Will It Blend?

Be wary of anyone trying to sell you a language as a silver bullet to solve every one of your development problems perfectly. No language is perfect, including IronPython. Even though the pros outweigh the cons, it's not fair to list only the good bits without addressing potential pitfalls.

- *Performance*: IronPython is an interpreted language and loses a bit of the performance that a compiled language will have.
- *Data visibility*: Everything in the Python language (and therefore IronPython) is considered public in terms of visibility. See "For Consenting Adults Only!" later in this chapter for some details on this point.
- *Semantics*: IronPython is a high-level language and therefore requires a bit of work on the interpreter's part to get it into a form the computer can use as instruction. As a result, some of the more complex aspects of programming are abstracted under simpler constructs; this does not degrade performance, but can simplify development efforts for the programmer.

■ **Note** High-level languages are considered easier for humans to read, whereas low-level languages are closer to machine instructions. IronPython code is so easy to read because of its high level. By contrast, if you were reading the machine code that the computer actually executes, it would be much harder to understand what's happening in the program. Although working with machine code is considered the most powerful way to program, it's also arguably the most difficult.

What Is a Dynamic Language?

I need to make an important point here before continuing: IronPython is a **dynamically typed language**. In dynamically typed languages, you do not have to define the type of a variable before you use it. The nature of IronPython variables and values may leave a sour taste in the mouth of programmers with a background in statically typed languages. Consider the C# snippet in Listing 1-1. After we look at a statically typed language and compare it to IronPython's dynamic typing, we'll examine exactly what makes up a dynamically typed language.

Listing 1-1. *A C# Method That Demonstrates Static Typing*

```
public void StaticTyping()
{
                // the value type on the left enforces a valid value➥
                assignment on the right
                int number = 5;
                string name = "Alan Harris";
                bool author = true;

                // the next line will throw a compile-time error;➥
               5 is not a valid boolean value
               bool yourName = 5;
}
```

Now compare this to an IronPython snippet that performs the same tasks (Listing 1-2), albeit with a slightly different result, which we will discuss afterward.

Listing 1-2. *An IronPython Method That Demonstrates Dynamic Typing*

```
def DynamicTyping():
                # the value type on the right dictates the variable type on the left
                number = 5
                name = "Alan Harris"
                author = True

                # this does NOT throw an error; the compiler➥
                infers the variable type from the value
                yourName = 5
```

See the difference? In a statically typed language such as C#, the compiler needs to know at compilation time what the type for a variable is. Attempting to assign a value type that does not match the variable type generates an error and the code won't compile. In the IronPython code in Listing 1-2, the variable called "yourName" is assigned a value of 5 because the compiler knows the desired type of the variable only by inferring from the type of the assigned variable. It is very important to be aware of this manner of assignment because it is in stark contrast to many other languages, but it is quite powerful if used correctly.

The eagle-eyed among you may have also noticed that when we started the method in C#, we included the word "void" to indicate that we would not be returning any value at the end of the method. If we wanted to return the value of the "name" variable, we would need to change the word "void" to "string" to tell the C# compiler that we want to return a value of data type string when we exit the method, and we would have to add "return name;" to the end of the method to avoid a compiler error and return the data type requested properly. This is not the case in IronPython; it will take care of handling this by examining the data type of the value returned when exiting the function.

There's something else going on here that was mentioned earlier in this chapter: you may notice that there is very little extraneous markup to indicate program flow. Missing are the line-ending semicolons, the method opening and closing brackets, and so on. IronPython relies on whitespace and indentation to control program flow (Listing 1-3).

Listing 1-3. *An IronPython Method That Demonstrates Program Flow via Whitespace and Indentation*

```
def DynamicTyping():
                # the value type on the right dictates the➥
                variable type on the left
                number = 5
                name = "Alan Harris"
                author = True

                # this does NOT throw an error; the compiler➥
                infers the variable type from the value
                error = 5
DynamicTyping()     # calls the method defined above and executes the assignments
```

We'll cover the nitty-gritty details of writing IronPython code shortly, but this should give you an idea of how this style of programming differs from other languages you may have used in the past. A lot of value is placed on readability in the Python (and therefore IronPython) dialect. Sacrificed are the brackets and the semicolons and embraced are

indentation and whitespace. This is a scary world for developers coming from C#, C++, or even VB .NET; there are no landmarks, no street signs by which to navigate. Rest assured, the benefits will become apparent quickly as you learn to develop applications rapidly, and as you learn how to work with IronPython these things will soon become second nature. There are pros and cons to dynamic typing, which are summarized in Table 1-1. As an IronPython developer you'll need to make informed choices about the trade-offs shown. We will see throughout this book examples where these considerations come into play.

Table 1-1. *Benefits of Static vs. Dynamic Typing*

Typing Style	Pros	Cons
Static	Enforced type safety at compile time, clarity of code, easier to debug, optimized machine code output	Rigid enforcement of type assignment, generally requires casting to change value types which hurts performance
Dynamic	Easier to write, allows execution of arbitrary code, simpler mocking during unit testing	Reduced type safety, increased potential for runtime errors, reduced code execution speed at runtime

What This Will Book Cover

Now that you've have taken the 10,000-foot tour of IronPython and its roots, this is as good a time as any to take the 10,000-foot tour of where we are going through the course of this book.

To encompass IronPython software development fully, we need to cover a lot of ground. Luckily we are .NET developers making use of a .NET dynamic language, so there is a lot of framework overlap between desktop and web applications, and what we learn in one situation applies to the other. We'll begin by looking at IronPython syntax and the different programming paradigms that an IronPython programmer can make use of. We'll then apply that knowledge to the creation of console applications. Console applications make a fantastic starting point for IronPython development because they allow us to focus on syntax without the complications of user interface design and the like. We won't stand on ceremony for long, though, because shortly afterward we'll delve into Windows forms applications and be making effective use of the CLR within IronPython code. Next we'll move onto database access and web applications, before finally wrapping things up with enterprise solution recipes, which will cover ways you can use IronPython in large-scale projects (regardless of whether they are legacy systems or being built from the ground up).

Although it is completely feasible that you learn solely by reading, I highly suggest that you take the time to work through and code each of the examples in this book. I would argue that it is more effective to learn by doing than by reading, and “real programmers” love to get their hands dirty. The projects and code are designed to be fairly quick activities to get you used to thinking in IronPython terms as well as to create a nice library of code for you to reference later. By the end of the book, you (ideally) will have quite a few folders of IronPython code at your fingertips.

Note In developer jargon, “real programmers” are just developers whom you might describe as fitting in the “hardcore” category, although the scale for measuring a real programmer seems to be a sliding one. It would be accurate to say that “real programmers” have an in-depth knowledge of the language and systems with which they choose to work. We’ll leave the arguments about using an IDE (interactive development environment) and language choices by the wayside.

Who This Book Is For

This book is designed for people new to Python and IronPython who want to get up to speed quickly and start learning the language with a minimum of fuss. I’m not going to cover every exhaustive language detail; instead I want to get you writing code in IronPython that can either stand by itself or work effectively with other applications that target the Microsoft .NET framework. I don’t assume you’ve got a computer science degree, nor do I assume this is your first time using a computer; rather I assume that I would best serve you by integrating some software design principles in the examples as we go so that your code is of high quality at the end. We will cover the language syntax, build from small components to larger applications throughout each chapter, and hopefully have a little fun doing it.

Although the beginning of this book is heavily slanted toward console applications (meaning boring old DOS prompts) and then Windows Forms applications, the ending of the book does cover web development in several ways. If you happen to have experience with XHTML and CSS, great! If not, no worries. I will give you a crash course as we go that should keep everything clear and understandable. You don’t need to be a web design guru to make it through those sections.

For Consenting Adults Only!

That sounds a lot scarier than it actually is, but I want to grab your attention on this one. “For consenting adults only” is a phrase that gets trotted out sometimes when Python is brought up in conversation, for some very good (if not tongue-in-cheek) reasons.

If you've already got some programming experience under your belt, you have likely encountered the notion that objects and data can be public, private, or some variety therein. In Python, everything is considered public, and the language won't do a thing to protect any objects or data you want to be private (the tongue-in-cheek humor being that you can always touch Python's private parts!). This is actually not the worst thing in the world, and we will cover ways to enforce good programming decisions and implementations. Just know that the Python language on which IronPython is based will let you make really, really boneheaded decisions "because we're all adults here."

If all this talk about public and private objects is alien to you, no problem! We'll cover everything in due time. It's just very important to remember that Python won't always cover your rear end, and a developer that is hip to that notion at the beginning is the wiser for it.

Prerequisites

There are a few requirements for installing and using IronPython, so let's make sure your system is set up properly. The great news is that to take advantage of everything in this book, you don't have to spend a cent. Every tool and component described next is free of charge, and I've noted where and why that is the case. I am using Microsoft Windows Vista on my machine, and my instructions and examples reflect that. But nothing in terms of the operating system is really any more complex than moving between folders and running programs, so it should be simple enough to follow along in your OS of choice.

- *Microsoft .NET 3.5*: The IronPython build relies heavily on new features available in the .NET 3.5 framework. Since I will be using Visual Studio 2008 throughout this book, you will need to download .NET 3.5 here: `http://www.microsoft.com/downloads/details.aspx?FamilyID=ab99342f-5d1a-413d-8319-81da479ab0d7&displaylang=en`.
- *Microsoft Visual Studio 2008*: When we reach the chapters in the book that involve interacting with other .NET applications, you will get the most benefit by having a copy of Visual Studio available. Microsoft has generously made available free editions of their Visual Studio suite, and we will be using Visual C# 2008 Express Edition throughout this book. It is available at `http://www.microsoft.com/express/vcsharp/`.

- *Microsoft SQL Server 2008*: The sections on database access assume that we are using SQL Server 2008 as our storage system. As with Visual Studio 2008, Microsoft has released a free edition of SQL Server 2008 that will serve us well later on. You can download this version at `http://www.microsoft.com/express/sql/default.aspx`. This installation is a bit more complex and requires a few configuration steps as well as an application reboot.

- *IronPython 2.0*: This is available as a free download; you'll need to get the latest release and install it. I have provided instructions here for how to do so. Although you are allowed to download the source binaries, for the purposes of this book we will deal with the compiled binaries and not concern ourselves with building or modifying IronPython itself.

DOWNLOADING AND INSTALLING IRONPYTHON

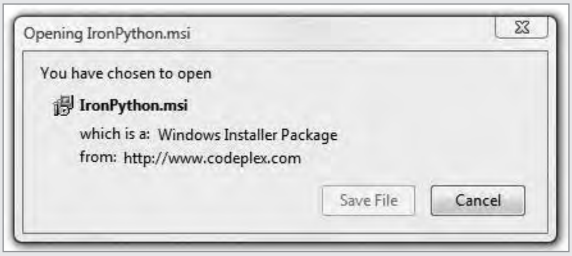
At the time of this book's writing, IronPython was at version 2.0 (December 10, 2008, release) and had a new home at CodePlex. The IronPython CodePlex site is your one-stop shop for downloads, tutorials, samples, and forum discussions.

1. In the web browser of your choosing, go to `http://www.codeplex.com/IronPython`.



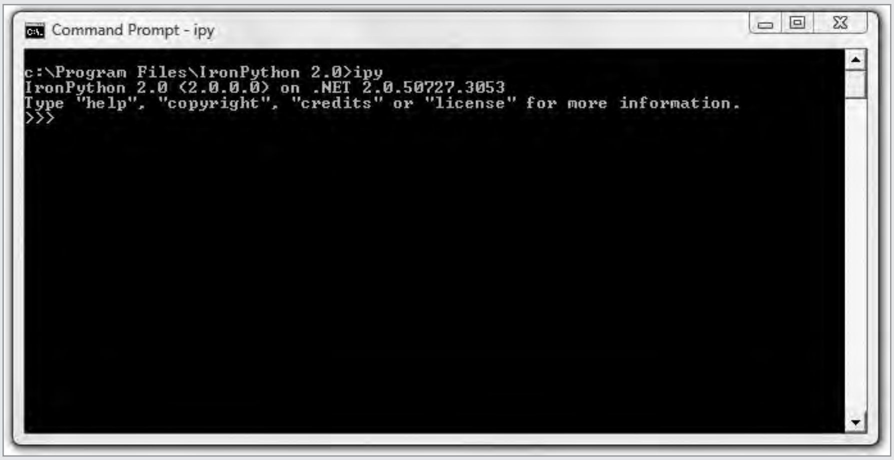
2. Click the Releases tab.

3. Click the IronPython.msi link, read and agree (if you do agree) to the License Agreement, and choose Save File when the file download window appears.



4. Once the file has downloaded, open the folder containing the Installer Package and run it. Follow the onscreen directions to install IronPython 2.0.

5. After the installation is complete, open a command prompt and go to the directory to which you installed IronPython. Type **ipy** and press Enter. You should see the following screen, which indicates that installation was successful.



IPY and You

What's this ipy.exe business all about? Quite simply, it's an interactive IronPython interpreter. You can execute IronPython code within this interpreter and see the results. Let's try it, shall we?

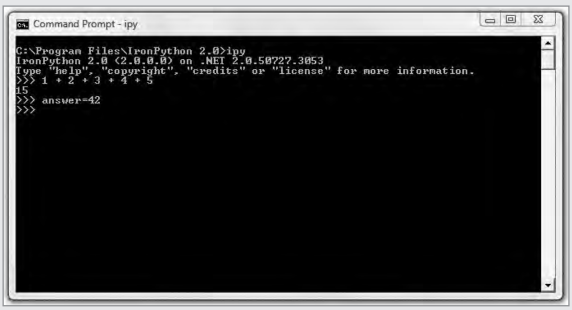
USING THE IRONPYTHON INTERPRETER

If you have not yet done so, open a command prompt and go to your IronPython directory. Type **ipy** and press Enter. The IronPython interpreter should start.

1. At the prompt (>>>), type **1 + 2 + 3 + 4 + 5** and press Enter.
2. The interpreter should return 15, followed by a new prompt.



3. Type **answer = 42** and press Enter. The interpreter should immediately respond with a new prompt.



4. Type **print "The ultimate answer to everything is", answer** and press Enter.



5. Type **exit()** and press Enter to exit the interpreter.

The interpreter is not only for typing code in real time. It can also run IronPython scripts, which are basically text files that contain specific instructions to make IronPython do what you want. We'll cover that usage immediately in Chapter 2 and continue to use the interpreter that way for a while. However, it's always a good idea to keep a command prompt open with the interpreter running while you are working; it's a quick and easy way to test certain things without having to run an entire application, for production applications tend to get very large.

■ **Note** If you're looking for a little splash of color in your life, type **ipy –X:ColorfulConsole** at the command prompt and press Enter. You should see color-coding instead of the plain monochrome format. You can check out the full range of interpreter command-line options by typing **ipy -?** at the command prompt.

Summary

IronPython is an exciting addition to the .NET language family. It is a powerful, dynamic language that allows developers to harness the power of the CLR within the Python syntax and to create applications that are easy to create, read, and maintain. We covered the history of Python and IronPython, discussed the benefits of using IronPython, and walked through installation and verification of IronPython from the CodePlex site before firing up the IronPython interpreter and solving the mysteries of the universe at the same time.

CHAPTER 2

IronPython Syntax

"High thoughts must have high language." — Aristophanes

Now that we've seen where IronPython comes from, it's time to start the fun stuff. Specifically we're going to start writing some IronPython code, learn how IronPython handles various programming constructs, and build the knowledge foundation for the rest of the book. Ready?

Data Types and Control Structures

At a very basic level, programming is all about manipulating data via instructions to the computer, with the goal of performing whatever tasks need to be completed, usually in a specific sequence. Since the core of this practice revolves around data, let's start by examining how IronPython handles different data types. After we've looked at some basic data types, we'll look at various control structures, which alter the way a program executes depending on various criteria (including input, raw data, and errors.)

A lot of truly fantastic programming books start off with examples of how to display "Hello World!" to the user, and I'm not about to break tradition.

Note Totally random trivia: the archetype of the Hello World! programs is actually *"The C Programming Language"* by Brian Kernighan and Dennis Ritchie. Since the book in your hands is about IronPython, which is based on CPython, which was written in and (in part) modeled after the C language syntax, it seems only fitting to keep the tradition alive.

Strings

Fire up your favorite text editor, type the block of code in Listing 2-1 exactly, and save it as HelloWorld.py in a convenient directory. For the sake of our examples in this book, I will be keeping my source code in C:\Python.

Listing 2-1. *Hello World!*

```
def HelloWorld():
        # create a string variable that holds Hello World! as content
        greeting = "Hello World!"
        print greeting

HelloWorld()
```

Now, open a console window and go to the directory where you have IronPython installed. Type **ipy c:\python\HelloWorld.py** and press Enter. If all goes well, you should see results like those displayed in Figure 2-1.

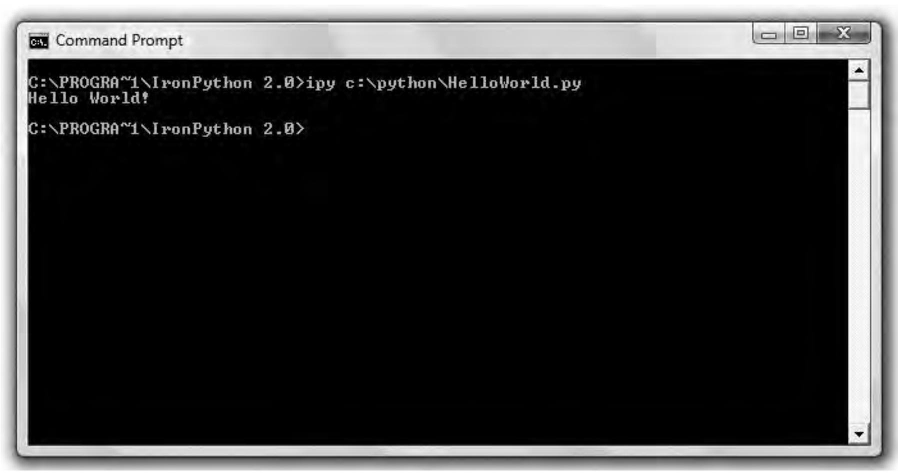


Figure 2-1. *Hello World, IronPython style*

Let's take a closer look at exactly what happens. We defined a method called *HelloWorld* that accepts no parameters. We added a comment for the benefit of ourselves and future maintenance programmers (which might be you too!) that describes why the following code is structured a particular way. Then we assigned a string value to a variable named *greeting* and proceeded to display it to the screen. The final line calls our HelloWorld method and executes our instructions.

Integers

Having tried strings, let's take a stab at integer values. As you probably recall from math class, integers are whole numbers that can be expressed without fractional or decimal components. For example, 1 is an integer, 1.1 is not. IronPython happens to function quite well as a calculator, so let's put it to work (Listing 2-2).

Listing 2-2. *Hello Math!*

```
def HelloMath():
            #create a handful of integer variables
            spam = 1
            eggs = 2
            print spam + eggs

HelloMath()
```

Run this program. Did you get 3 as output? If so, great! If not, correct any errors in syntax so that your program matches Listing 2-2.

Let me reiterate at this point that IronPython is a dynamically typed language. Nowhere in this code did we indicate variable types; the compiler figures that out based on what appears to the left side of the assignment operator (the equals sign.) Don't just take my word for this. Let's prove it. Listing 2-3 does just that.

Listing 2-3. *Dynamic Typing at Work*

```
def HelloDynamic():
            # create a string variable that holds Hello World! as content
            greeting = "Hello World!"
            print greeting

            #create a handful of integer variables
            spam = 1
            eggs = 2
            print spam + eggs

            #print our greeting, followed by the sum of spam and eggs
            print greeting + spam + eggs

HelloDynamic()
```

Run this program. Did it print out *Hello World!*, then *3*, and then *Hello World! 3*? It didn't? If you're playing along at home, what you probably just saw was the compiler rudely telling you that something went wrong. The error looks something like Figure 2-2.



Figure 2-2. *IronPython complains quite loudly when it encounters a problem.*

This is dynamic typing in action. We have just instructed the compiler to add *Hello World!*, the number *1*, and the number *2* together. It doesn't make sense, does it? That's what IronPython thinks too! We need to tell IronPython that we want to concatenate these things and to display everything to the screen. We'll go ahead and modify the code in Listing 2-3 and see if we can give IronPython the instructions it needs to understand what we want (Listing 2-4).

Note In programming terms, *concatenation* refers to joining two or more strings together. One easy way to think of this would be your own name. Frequently registration forms offer fields for both your first name and your last name separately, but when you log in the application says something like, "Welcome back, Alan Harris!" It does this by concatenating, or joining those strings together to create one string.

Listing 2-4. *Dynamic Typing at Work, Fixed!*

```
def HelloDynamic():
        # create a string variable that holds Hello World! as content
        greeting = "Hello World!"
        print greeting
```

```
        #create a handful of integer variables
        spam = 1
        eggs = 2
        print spam + eggs

        #print our greeting, followed by the sum of spam and eggs
        print greeting, spam + eggs

HelloDynamic()
```

```
Hello World!
3
Hello World! 3
```

Success! The use of the comma instead of the plus sign tells IronPython that first we want to display the string value of the greeting variable and then we want to display the integer result of adding spam and eggs together. Again, if you do not see the output I described, check your syntax against the code in Listing 2-4 and make sure everything is correct.

Conditional Statements

Now that we have a resounding victory under our belts, we need to throw a wrench in the works. It's a very rare program "in the wild" that operates from top to bottom without any change in application flow. More often than not, certain code is executed based on a given condition and other code is not. How does IronPython allow us to handle those situations? The answer is a **conditional statement**. A conditional statement allows us to set a criterion (or condition) that alters the way a program executes. We're also going to add a dash of user input to this program so that the user has a say in how the program executes (Listing 2-5).

Listing 2-5. *Getting User Input*

```
def HelloConditional():
    firstName = raw_input("Please enter your first name: ")
    lastName = raw_input("Please enter your last name: ")
    age = int(raw_input("Please enter your age: "))
```

```
    # change the program output based on the user's age
    if age < 25:
            print "You, ", firstName, lastName, ", are younger than the author."
    elif age == 25:
            print "You, ", firstName, lastName, ", are the same age as the author."
    else:
            print "You, ", firstName, lastName, ", are older than the author."

HelloConditional()
```

```
Please enter your first name: Alan
Please enter your last name: Harris
Please enter your age: 25
You, Alan Harris , are the same age as the author.
```

Depending on what you entered for your age, you may see that you are younger or older than I or perhaps that you are the same age. But the key here is that the output of the program, indeed the entire program operation itself, *changed* based on this conditional statement. You may have noticed that I slipped two new concepts in there without telling you. You'll see that I made use of the *raw_input()* function and that when I asked for the user's age, I wrapped the entire input line in *int()*. Wrapping the entire input line in *int()* is an operation known as **casting**. As I just discussed, the *raw_input()* function takes a string value for input. So without the integer cast, the age variable would contain a value like "25" instead of 25. As we'll see in just a moment, it's going to be very important to have the correct data type because we need to make some comparisons, and in the eyes of the computer the string value "25" is not equal to the integer value 25.

■ **Note** *Casting* is a method of converting between two data types. In the preceding example, we were receiving input in the form of a string, but we converted that to an integer. You could also do the reverse and convert a string back to an integer. There are a wide variety of types to convert between, but not every conversion works; for example, you can convert the string "1234" to an integer with a value of 1234, but you can't convert "My name is Fred" to an integer.

Input() or Raw_Input()

The *raw_input()* function is a safe method for getting input from the user because it does not allow arbitrary execution or evaluation of user input. There is another, similar command in IronPython called *input*. The *input* command, while useful, can also be very dangerous; it allows the execution of IronPython commands based on user input. This can be a tremendous security risk and should be used with caution. Allowing your users to execute any IronPython code they see fit should always raise a red flag in your mind. The *raw_input()* function is more appropriate because it accepts a string input that cannot be executed directly. We'll cover use of the *input* function later, but for right now consider it a best practice to stick to *raw_input()* for getting input from the user.

■**Note** A very close relative to this concept is that of SQL injection. SQL stands for Structured Query Language, and it's a very common way to communicate with a database. One of the simplest and certainly most overlooked security flaws is neglecting to secure communication to and from the database. Much like executing arbitrary IronPython commands is a security risk, not protecting your SQL commands is a terrible security hazard. We'll cover SQL and protection against injection attacks in Chapter 7.

Error Handling and Exceptions

From the heading of this section, some of you may have already guessed about one glaring omission from this program. I'll spoil it for the rest of you: **error handling**. This program works great if you provide it exactly what it needs in exactly the format it expects. But run it again; this time enter **-1** as your age. It should output something like the following.

```
Please enter your first name: Alan
Please enter your last name: Harris
Please enter your age: -1
You, Alan Harris , are younger than the author.
```

Now technically, that is correct; someone who is –1 years old is definitely younger than I am. But when is the last time you met someone who was younger than 0 years? This is an example of a **logical error**, that is, an error that does not stop the program from continuing execution but that results in flawed or inaccurate data or output. Logical

errors are generally far harder to identify and correct than **syntax errors**, which are errors in the source code instructions themselves. We need to write some validation rules that ensure that the input we're getting from the user meets the criteria we need to guarantee proper program operation. Rules like the ones we're about to create typically exist in the **business layer** of an application. But we're not quite that far along yet, so for right now we'll place it alongside the input code.

■ **Note** You may hear developers talking in feverish terms about presentation, business, and data layers in their programs. Typically this refers to the traditional three-tier application design. *Tier* ordinarily denotes a physical separation of components, whereas *layer* is more of an abstract concept, but the terms tend to be interchangeable in conversation. The *presentation layer* is generally the user interface, the *business layer* handles all rules related to the business or application domain (as well as input validation and providing the presentation layer with content to display), and the *data layer* takes care of speaking to and retrieving data from your database. We'll discuss these concepts in depth later; I just wanted them on your radar for now and to get you thinking about ways to improve our growing application design.

How can we best ensure that we have handled this particular error gracefully? It makes sense to compare the user's input to some known sanity checks, such as "a user cannot have a negative age" or "a user cannot be more than 150 years old." Always be very careful when designing rules like these; guaranteed you'll choose 150 as the maximum age for a user to input, only to find that somebody out there is not only 151 years old, but 151 years old and not happy that he can't use your program.

That said, let's modify our program to use the two rules we on which we just decided . Note that this is not a comprehensive rule set that covers every possible error. We need to start small and then identify trouble spots and how to fix them. Listing 2-6 adds a touch of error handling to the application.

Listing 2-6. *Getting User Input with Error Handling*

```
def HelloConditional():
        # get a few data values from the user
        firstName = raw_input("Please enter your first name: ")
        lastName = raw_input("Please enter your last name: ")
        age = int(raw_input("Please enter your age: "))
```

```
        if age < 0:
             print "You, ", firstName, lastName, ", need to input a valid age➥
before continuing!"
             HelloConditional()
        elif age > 150:
             print "You, ", firstName, lastName, ", need to input a valid age➥
before continuing!"
             HelloConditional()
        elif age < 25:
             print "You, ", firstName, lastName, ", are younger than the author."
        elif age == 25:
             print "You, ", firstName, lastName, ", are the same age as the author."
        else:
             print "You, ", firstName, lastName, ", are older than the author."

HelloConditional()
```

```
Please enter your first name: Alan
Please enter your last name: Harris
Please enter your age: -1
You, Alan Harris , need to input a valid age before continuing!
Please enter your first name:
```

That's great! Now our program has some knowledge about what constitutes a valid range of inputs. But it can still be tripped up. Run the program again and provide some decimal value (such as 25.1) for your age. You will see something like the following.

```
Please enter your first name: Alan
Please enter your last name: Harris
Please enter your age: 25.1
Trackback (most recent call last):
File 'c:\python\HelloConditional.py', line 20, in c:\python\HelloConditional.py
File 'c:\python\HelloConditional.py', line 4, in HelloConditional
ValueError: invalid integer number literal
```

Yikes! We didn't even get asked for our name and age again. It just crashed and burned! This is an example of an **exception**. An exception occurs when program execution strays far off course. A pretty common example would be dividing by 0. Many

languages (C#, for example) even have built-in DivideByZero exceptions to handle those types of cases. In fact, IronPython just told you what type of exception it threw when it encountered the error: a ValueError exception. We can check for those types of exceptions and handle them accordingly.

Try-Catch-Finally

Wouldn't it be nice if we could just wrap potentially unsafe operations in some sort of code block that could notify us if things go terribly awry? IronPython provides a construct that will suit this purpose perfectly, but with a few caveats, which we will address after looking at an example (Listing 2-7).

Listing 2-7. *Getting User Input with Error Handling and Exception Handling*

```
def HelloConditional():
        # get a few data values from the user
        firstName = raw_input("Please enter your first name: ")
        lastName = raw_input("Please enter your last name: ")
        try:
             age = int(raw_input("Please enter your age: "))
        except ValueError:
             print "You, ", firstName, lastName, " need to input a valid age➥
before continuing!"
             HelloConditional()
        if age < 0:
             print "You, ", firstName, lastName, ", need to input a valid age➥
 before continuing!"
             HelloConditional()
        elif age > 150:
             print "You, ", firstName, lastName, ", need to input a valid age➥
before continuing!"
             HelloConditional()
        elif age < 25:
             print "You, ", firstName, lastName, ", are younger than the author."
        elif age == 25:
             print "You, ", firstName, lastName, ", are the same age as the author."
        else:
             print "You, ", firstName, lastName, ", are older than the author."

HelloConditional()
```

```
Please enter your first name: Alan
Please enter your last name: Harris
Please enter your age: 25.1
You, Alan Harris , need to input a valid age before continuing!
Please enter your first name:
```

Now that's a lot more user-friendly. We have wrapped the potentially offending code in what is known as a **try-catch block**. In a *try-catch* block, we first "try" to execute the code within the *try* section. If an exception is "thrown," we *catch* it and perform some action. There may be situations in which you want a block of code to execute at the end of the block, regardless of whether or not an exception is raised. For those situations, there is an additional step, making what is termed a **try-catch-finally block**. If you add a *finally* section, the code within is guaranteed to execute immediately upon exiting the *try* block. Later, when we delve into more advanced concepts, we'll learn that the *try-catch-finally* block is critical to releasing valuable system resources that otherwise may become bogged down with unhandled exceptions, leading to memory leaks and other nastiness. Listing 2-8 adds a *finally* block to the exception handlers.

Listing 2-8. *Exception Handling with a Finally Block*

```
def HelloConditional():
        # get a few data values from the user
        firstName = raw_input("Please enter your first name: ")
        lastName = raw_input("Please enter your last name: ")
        try:
             age = int(raw_input("Please enter your age: "))
        except ValueError:
             print "You, ", firstName, lastName, ", need to input a valid age➥
before continuing!"
             HelloConditional()
        finally:
            print "This code is executed in the finally block."

        if age < 0:
            print "You, ", firstName, lastName, ", need to input a valid age➥
before continuing!"
            HelloConditional()
        elif age > 150:
             print "You, ", firstName, lastName, ", need to input a valid age➥
before continuing!"
```

```
            HelloConditional()
        elif age < 25:
            print "You, ", firstName, lastName, ", are younger than the author."
        elif age == 25:
            print "You, ", firstName, lastName, ", are the same age as the author."
        else:
            print "You, ", firstName, lastName, ", are older than the author."

HelloConditional()
```

I should stress the importance of not structuring your program execution solely around exceptions. There is a difference between handling exceptions and validating data entry. We have conditional statements in Listing 2-8 to handle direction of the program under normal circumstances. A user's entering the string "twenty-five" for his or her age is not a normal circumstance by our definition, so we have an exception to handle it. Exceptions are "heavier" than conditional statements and have a slightly larger impact on performance. By definition, they should be exceptional cases, not the way you control program flow. Bear that in mind as you develop your code.

Built-In Functions

Occasionally throughout this chapter I've made references to built-in IronPython functions, such as *raw_input()*, but I haven't really broken down what these functions are or how to find them. IronPython includes quite a few built-in functions (current to version 2.5.2 of CPython as of the time of this writing), which I will list next. Then we will walk through some of the more frequently used ones and see some examples. You can learn more about the other functions at `http://www.python.org/doc/2.5.2/lib/built-in-funcs.html`. For the remaining portions of this chapter, I will assume that you have an IronPython interpreter open. But you don't need to make any .py files; we'll just use the built-in interpreter for now.

abs

For various reasons, you may need the absolute value of a number. You could do something like sign checking and then multiplying by –1 to flip that sign, but that's unnecessary! IronPython gives you the *abs()* function for convenience.

```
>>>  print abs(-5)
5
```

chr

If you are working with a bit of code where the individual letters in a string of characters have been converted to their ASCII values, you may need to convert them back. The *chr()* function is meant to do that. It is something of a sister function to the *ord()* function, which we will take a look at shortly. Feel free to flip forward to the *ord()* function description and example and to return here.

```
>>>  print chr(65), chr(108), chr(97), chr(110)
A l a n
```

Note ASCII values refer to a range of integer values, from 0 to 255, that can be used to represent a variety of characters, including letters, numbers, symbols, and so on. Luckily, you don't have to remember them all. There are quite a few handy tables out there that list the ASCII values from 0 to 255 as well as various conversions (hexadecimal, octal, etc.). I use `http://www.asciitable.com` frequently for that purpose.

---

dict

We're going to look at sets and dictionaries in more depth in the next chapter. The *dict()* function allows you to specify a *dictionary*, which is a collection of data that is enumerated in key-value pairs. Note that each key must be unique but that values can repeat. In the following example, the keys are the names of the colors and the values are the numbers assigned to them. You will notice that red and yellow are assigned the same value.

```
>>>  dict(red=1, blue=2, green=3, yellow=1)
{'red' : 1, 'blue' : 2, 'green' : 3, 'yellow' : 1}
```

In this code output we can clearly see IronPython showing us the key-value pair configuration.

Now let's see what happens if we try to duplicate key names in the same dictionary.

```
>>> dict(red=1, blue=2, green=3, red=2)
  File "<stdin>", line 1
    dict(red=1, blue=2, green=3, red=2)
                                 ^
SyntaxError: duplicate keyword argument
```

IronPython didn't care for that at all. The error tells us there was a problem with a duplicate key in line 1 of *stdin*, which means "standard input," in this case the keyboard. In short, we messed up! Remember to keep keys unique in your dictionaries. Later in this chapter we discuss the *list()* function, which has some very important differences from the *dict()* function. When you learn more about lists, refer back to this for the sake of comparison.

dir

When we cover importing modules, in the next chapter, you may find it useful to enumerate the names of modules available in the symbols table. One of the first modules we import is the **sys** module, so let's compare the output of the *dir()* function before and after running an import.

■ **Note** Importing .NET code modules in IronPython is handled very similarly to how it's handled in traditional Python, with the notable exception of needing the *clr* module to access the .NET namespaces. From a programming standpoint, the effect is the same.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import sys
>>> dir()
['__builtins__', '__doc__', '__name__', 'sys']
```

So we can see that before we imported an additional module, the *__builtins__*, *__doc__*, and *__name__* modules were available to us. These provide basic IronPython functionality. We imported the *sys* module and it became **enumerated** in our list. We can drill down further to see what functions the *sys* module provides by running the *dir()* function and providing the module name as a parameter.

■ **Note** *Enumeration* is the act of assigning a unique numerical value to an object in a list. A Social Security number is a type of enumeration; it establishes a unique identity for a specific entity. We'll cover enumerations a bit more in Chapter 6 when we build a plug-in system using C# and IronPython.

```
>>> dir(sys)
['__name__', '__stderr__', '__stdin__', '__stdout__', '_getframe', 'api_version'
, 'argv', 'builtin_module_names', 'byteorder', 'copyright', 'displayhook', 'exc_
clear', 'exc_info', 'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exe
c_prefix', 'executable', 'exit', 'getcheckinterval', 'getdefaultencoding', 'getf
ilesystemencoding', 'getrecursionlimit', 'hexversion', 'maxint', 'maxunicode', '
meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform',
'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setrecursionlimit', 'settrace', 'st
derr', 'stdin', 'stdout', 'version', 'version_info', 'warnoptions', 'winver']
```

These are the various functions that *sys* exposes to you as a developer. The *dir()* function is very useful for listing classes in a module

Files via open

Files I/O operations are used when you want to interact with physical files on the computer. Perhaps you are running a web crawler that you created and you want to save a list of all the URLs on a website to a file on your drive. Files can be opened using the *open()* command, with the location of the file on disk provided as a parameter. In the following example, the response from the interpreter tells you that the file you asked for exists and has been opened in "read" mode.

```
>>> open("c:\python\HelloWorld.py")
<open file 'c:\python\HelloWorld.py', mode 'r' at 0x00E441DF>
```

File operations are very, very expensive compared to operations in memory. For example, adding two integers is considerably slower when you have to open two text files from disk to get those integers in the first place. You may not perceive much of a difference when opening one or two files on your own machine, but trust me when I say that under load it's a very heavy type of operation to complete.

■ **Note** If ever a family of operations deserved to be wrapped in *try-catch-finally* blocks, file operations are those. Beside the fact that any sort of communication to and from the drive is very costly in terms of performance, a critical failure that leaves disk resources open is one of the fastest ways to leak resources on a system.

---

for (iterations)

Suppose for a moment that you had a list of students in an elementary school class and that you wanted to produce a list of names for the teacher to take attendance. How would you do that? In IronPython, you can **iterate** over a list of items using what is called a **for loop**.

■**Note** There are many types of looping structures in programming, such as *while*, *do while*, *do until*, *for*, and so on.

When you iterate over a list, what you're telling IronPython to do is go step-by-step through a list of items and allow you to perform some task on the item or data. In the following example, we're going to create a list of students, and then we are going to iterate over the list and display each student on a separate line of output for the user.

■**Note** A list of items in IronPython is expressed as an array. *Arrays* are in the format *variableName = ['value1', 'value2', 'value3']* and so on.

```
>>> students = ['Susie', 'Bobbie', 'Tommy', 'James', 'Harry', 'Sally', 'Larry',➥
'Moe', 'Curly']
>>> for student in students: print student
… [Press 'Enter' key]
Susie
Bobbie
Tommy
James
Harry
Sally
Larry
Moe
Curly
```

The *for* loop is constructed in such a way that we need to provide an object for each individual item to live in so that we can do some work on it. When we created the *for* loop we told it to create an object called "student" for each item in the list. Our instruction

could be read, "for each element in the list students, take that element and place it in a student object, then execute any code following the colon symbol." So for each element in the students list, the value of the student object (in this case it will be the student's name) is printed to the screen. Control flow returns to the loop to iterate to the next item in the list, if there are any. If not, the loop exits.

help

Sometimes you get stuck! IronPython is there to help. Similar to the *dir()* function, you can ask for help either with or without a function in mind.

■ **Note** For the sake of brevity I have not included the entire output of the *help()* function, only examples of the command itself.

```
>>> help()

>>> help('print')
```

hex

As developers, we may have to interact with a variety of numerical systems. Most of us are quite used to dealing with decimal numbers, which are base-10 numbers (i.e., 0, 1, 2, 3, . . . , 10, 11, 12, 13, . . . , etc.). Computer and software systems often use hexadecimal numbers, which are in base-16. Calling these hexadecimal *numbers* seems a bit of a misnomer, because they actually involve letters as well. For example, whereas the decimal representation of *thirteen* is 13, in hexadecimal it would be D. How is that possible? It seems silly, but think of the fingers on your hands. Assuming you have all 10 of your fingers, you could start at the far left, with your pinky as 0, and count all the way to your other pinky and end up at 9 (0 through 9 being a total of 10 numbers.) That's base-10. What if you were to add six fingers to the end of your right hand? Besides looking a bit odd, you'd have to have some way of referring to them numerically. Mathematicians chose letters, so those next fingers would be A, B, C, D, E and F.

IronPython makes converting from decimal to hex very simple. Take a look.

```
>>> hex(13)
'0xd'

>>> hex(123)
'0x7b'
```

What's with the 0x before the hex values? It's just a way of representing hex values that has been carried over from C, which is the language in which CPython was created. Since IronPython is built with CPython at its core, the convention continues.

■ **Note** Remember the ASCII values we described earlier that ranged from 0 to 255? There are hexadecimal conversions of each of those numbers, ranging from 0 to FF. Without cheating (I'm looking at you, Google!), see if you can work out how 255 equals FF.

---

int

Similar to hex, we may need to extract the integer value from a decimal number. You can pass a number to the *int()* function, and the integer portion of the value will be returned.

```
>>> int(123.456)
123

>>> int(9018724.8971230987)
9018724
```

len

If you need to count the number of items in an object or sequence quickly, the *len()* function is your best bet. With it you can count the number of objects in a dictionary, a list, or an array, the number of characters in a string, and so on.

```
>>> len('spam_and_eggs') # note that the quote marks are➥
not counted as part of the string length
13

>>> colors = dict(red=1, blue=2, green=3)
>>> len(colors)
3
```

list

Earlier in this chapter we looked at dictionary objects, which are key-value pairs where the key must be unique. Lists are similar to dictionaries, except they are not key-value

based and are therefore free of the restriction that keys must be unique. Lists can also be expressed without a function keyword.

```
>>> colors = ["red", "blue", "green", "red", "blue", "green"]
>>> list(colors)
['red', 'blue', 'green', 'red', 'blue', 'green']
>>> list('The quick brown fox jumped over the lazy dog')
['T', 'h', 'e', ' ', 'q', 'u', 'i', 'c', 'k', ' ', 'b', 'r', 'o', 'w', 'n', ' ',➥
 'f', 'o', 'x', ' ', 'j', 'u', 'm', 'p', 'e', 'd', ' ', 'o', 'v', 'e', 'r', ' ',
 't', 'h', 'e', ' ', 'l', 'a', 'z', 'y', ' ', 'd', 'o', 'g']
```

It's important to understand the difference between dictionaries and lists, and choosing the correct one for the type of operation you're doing is critical. It's hard to say that one will work better for you in all or most cases; neither is truly superior to the other. There are situations where a dictionary is the way to go and situations where a list would be a better choice. In general, if you need something in the collection of items to be guaranteed unique, choose a dictionary. If you have a collection of items where nothing needs to be unique and items can be repeated without any problems, a list would be an appropriate solution.

max and min

In any set or list, you may have an interest in finding the largest or smallest element quickly. In some cases, you may choose to create your own functions to perform these tasks for performance or other reasons, but in most cases you can rely on the built-in *max()* and *min()* functions. These functions work on strings, numbers, dictionaries, and other types of objects.

```
>>> max('spam_and_eggs')
's'
>>> min('abcdefghijklmnopqrstuvwxyz')
'a'
```

ord

The *ord()* function is the counterpart to the *chr()* function we discussed earlier. Where you would use *chr()*to get the character equivalent from a numeric ASCII value, you would use *ord()*to get the numeric ASCII value from a character.

```
>>> print ord('A'), ord('l'), ord('a'), ord('n')
65 108 97 110
```

Note The *ord()* function will work only on single characters, not strings. You cannot pass ('Alan') as the parameter; you can only pass one character at a time or you will receive an error telling you something to that effect ("expected a character, but a string of length *n* was found.")

pow

Remember the quadratic formula from elementary and high school? (It's $ax^2 + bx + c = 0$, for those who need the refresher!) The first term of the polynomial, ax^2, is read as "variable *a* multiplied by *x* raised to the second power." It's that word *power* we're interested in here. The *pow()* function is used to raise a number *n* by power *p*, or *n* multiplied by itself *p* times. It is used in a format of pow(*n, p*), where *n* and *p* are numeric values, such as integers or floats.

Note We haven't yet covered working with floating-point values; for right now the short explanation is that they are numbers that need to be expressed with a decimal component. The number 10 is an integer and does not need the decimal or any additional values after the decimal to express its complete value; the number 10.12345 is a floating-point number and needs the decimal and additional values to be expressed properly.

```
>>> pow(1, 2)
1
>>> pow(2, 2)
4
>>> pow(3, 3)
27
>>> pow(3.14159, 2)
9.869587728099999
```

This function is great for stuff like finding the area of a circle, which is equal to the value of pi times the radius of the circle squared ($A = \varpi r^2$). You can compute the area of a circle with code such as the following.

```
>>> import math
>>> radius = 5
>>> area = math.pi * pow(radius, 2)
>>> print area
78.5398163397
```

Note We made use of the built-in math module, which we haven't covered yet. The math module has a wide variety of mathematical operations included so that you don't have to take the time to implement them yourself. For this example we used the *math.pi* function to get an approximation of the value for pi, which in IronPython is equal to 3.141592653589793. If you don't want to use the math module, you can use that approximation to get the same results.

random

I'm going to take a little liberty as the author at this point and employ a function that is technically built in but that requires you to import a module to use it.

As a programmer you'll find plenty of situations where you want to generate random numbers. You may need them for logic decisions in a computer game, to create dummy data for testing code or algorithms, and so on. Generating totally random numbers is difficult. Technically, most random number generators are not truly random; they are termed *pseudo-random*. The random number generator in IronPython is indeed a pseudo-random generator, but it will suffice for many applications. To use it, you'll need to import the aptly named *random* module first, as in the following example.

```
>>> import random
>>> print random.random()
0.686993518233
>>> print random.random()
0.004263442943
>>> print random.random()
0.872601615671
>>> print random.random()
0.501539671096
>>> print random.random()
0.113435200468
```

Note Discussing random numbers and the many ways one can derive random numbers for a given system is both beyond the scope of this book and a rabbit hole I have no desire to travel down at the moment! It is important that I mention at this point that the random number generator in IronPython is *NOT* suitable for cryptography, which covers areas such as encryption. The random number generator is deterministic; if you were to observe a little more than 600 random numbers generated in IronPython, you would be able to determine all future outputs from those values alone. So if cryptography or security is in your development future, you will need to find alternative solutions. If you're interested in learning more about how IronPython generates random numbers, the algorithm it uses, called *Mersenne Twister*, is very effective at quickly generating pseudo-random numbers.

randrange

This function is almost identical in purpose to the *random()* function in IronPython, except it lets you be a bit more specific about the range and type of values you need. You may find yourself in a situation where you need a random integer between 1 and 52 (for a card game, perhaps.) The *randrange()* function will return an integer between values *n* and *m*. There is a comparable function that returns floating-point values called *uniform()*, which is discussed later in this chapter. Please note that *random()*, *uniform()*, and *randrange()* are defined in the *random* module, so you need to import the *random* module before using any of these functions.

```
>>> import random
>>> print random.randrange(1, 52)
21
>>> print random.randrange(1, 10)
5
>>> print random.randrange(-57, 3)
-15
```

round

Certain mathematical operations require you to round numbers. For example, in rounding to zero decimal places, 1.2 would round down to 1.0, and 45.9 would round up to 46.0. Note that although the answer returned from the *round()* function is a floating-point value, the number itself can be expressed without any digits following the decimal point (1.0 is equivalent to 1).

```
>>> round(1.2)
1.0
>>> round(45.9)
46.0
>>> round(75198.098123750)
75198.0
```

You can also provide a second parameter to the *round()* function that dictates how many digits after the decimal point you want to consider when rounding. By default, this is zero, but it can be whatever you like. Note that if you perform rounding with additional decimal points specified, the resulting answers may not be expressible as integers. See the following example for an illustration of this point.

```
>>> round(75198.098123750) # the answer can be expressed➥
 as an integer with no loss of precision (75198)
75198.0
>>> round(75198.098123750, 1) # this cannot; converting to an➥
integer would lose the decimal values
75198.1
>>> round(75198.098123750, 5) # we have specified 5 digits➥
after the decimal point (.09812)
75198.09812
```

I can hear someone out there now saying, “Aha! But how is .5 rounded? What happens if you’re exactly halfway between one number and another?” Well, there’s a simple answer for that. In IronPython, the rounding answer will always move away from 0 toward the next-largest number. This is very important to understand. A variety of rounding methods are in use today, and if you are not aware of how numbers are rounded in different systems you may encounter unusual behavior or results and not have any idea of where the fault lies (this is particularly true in e-commerce and banking software).

```
>>> round(0.5)
1.0
>>> round (-11029835019243.5)
-11029835019244.0
>>> round (592873.5)
592874.0
```

uniform

This function is almost identical in purpose to the *randrange()* function discussed earlier in the chapter, except whereas *randrange()* will return an integer value between values *n* and *m*, the *uniform()* function will return a floating-point value between values *n* and *m*. Please note that, like *random()* and *randrange()*, *uniform()* requires you to import the *random* module before using it.

```
>>> import random
>>> print random.uniform(1, 100)
81.2660990526
>>> print random.uniform(-10, 10)
0.907017286358
>>> print random.uniform(-200, 80)
-137.981091038
```

But Wait, There's More!

This is not an exhaustive description of every command and control structure in IronPython, just a general overview to get you comfortable with the general way IronPython code is structured and to beat you about the head and shoulders with the coding implications of working in a dynamically typed language. In an ideal world, coding would be simple and we'd be working with straightforward code such as what we've worked with so far. Unfortunately the world is far from ideal, and business problems sometimes require very complicated solutions.

■ **Note** You can find the complete list of built-in commands at `http://www.python.org/doc/2.5.2/lib/built-in-funcs.html`. In general, what's been listed in this chapter will allow you to get the job done and solve most programming problems, but it's always nice to have a complete, consistently updated resource. The underlying Python language is always evolving, and IronPython with it.

As we add more and more aspects of the IronPython language to your toolkit, we'll look at ways to keep applications easy to code and maintain so that even when they are thousands of lines long you'll be able to work efficiently with them. Always keep in mind that you may not be the person who maintains your code in the future. And even if you are, your memory may play tricks on you. A poorly implemented program is a nightmare to maintain—for anyone.

Summary

IronPython (and Python in general) is designed to be a very easy-to-read, easy-to-write language. In this chapter we explored the basics of IronPython syntax, including string and integer manipulation, error handling, and exception handling. We took a very simple Hello World application and added user input, string concatenation, and error checking to it. This is really the basic context in which a developer works: get input , work with that input to produce a desired output, and handle any errors encountered gracefully. We then took a closer look at the various built-in functions that IronPython provides to make your development life simpler and to prevent you from having to reinvent the wheel. By now you should have a pretty good grasp of how IronPython code is structured, and you should be comfortable with basic programming operations.

CHAPTER 3

Advanced IronPython

"Mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true." — Bertrand Russell

At this point we've already written some basic IronPython code. We've gotten input from the user, done some processing of that input, handled errors and exceptions gracefully, and provided output back to the user in a structured fashion. But we have only scratched the surface of working with IronPython. In this chapter we will expand our knowledge of strings, integers, floats, and other data types. We will also be introduced to object-oriented programming, and in the end we will build an object-oriented IronPython business solution. Welcome to Advanced IronPython!

String Operations Revisited

The output we have displayed to the user so far has been fairly simplistic. It has consisted of a few words put together into a line or two with no real concern for formatting or special characters. But certain very common situations require more complex output formats. For instance, suppose we want to create an application that takes the user's name as input and prints it back to the screen. Sample code to do this is presented in Listing 3-1.

Listing 3-1. *Returning User Input to the Screen*

```
def MyName():
     firstName = raw_input("Please enter your first name: ")
     lastName = raw_input("Please enter your last name: ")

     print "Hello, ", firstName, lastName, "."
MyName()
```

This is an acceptable solution. But suppose we decide to get fancy with our output. For example, if we want to put the user's name in quotation marks, we need a way to tell IronPython that we do not want to end the current string. This requires us to **escape** the

quotation mark. In IronPython we "escape" the quotation mark with a backslash, to indicate that it is a literal character (Listing 3-2).

■ **Note** In programming lingo, *escaping* a string can have multiple meanings. In this case, what we're doing is informing IronPython that we want to insert a specific character that the compiler normally interprets as being special. The term also comes up in web development when certain HTML characters need to be escaped to be interpreted properly, and it comes up as well in database development, where user-provided data is escaped for security purposes.

Listing 3-2. *Returning the User's Name with Quotation Marks Included*

```
def MyName():
     firstName = raw_input("Please enter your first name: ")
     lastName = raw_input("Please enter your last name: ")

     print "Hello,", "\"", firstName, lastName, "\"."

MyName()
```

```
Please enter your first name: Alan
Please enter your last name: Harris
Hello, " Alan Harris ".
```

The escape method has a variety of uses that are not limited to getting quotation marks where we need them. It can also function to introduce newlines if you need to move output down to a different line (Listing 3-3).

Listing 3-3. *Returning the User's Name with Quotation Marks Included and a Newline Character*

```
def MyName():
     firstName = raw_input("Please enter your first name: ")
     lastName = raw_input("Please enter your last name: ")

     print "Hello,", "\"", firstName, lastName, "\".\nIt is nice to see you again!"

MyName()
```

```
Please enter your first name: Alan
Please enter your last name: Harris
Hello, " Alan Harris ".
It is nice to see you again!
```

IronPython is also quite adept at pulling one or more characters out of a string using very little code. Remember that IronPython is meant to be compatible with CPython, which is itself based on the C programming language. Like C and CPython, IronPython arrays (be they character arrays or any other type) are zero-based, meaning that the first element in an array of *n* elements is the number 0. This is demonstrated in Listing 3-4.

Listing 3-4. *Demonstrating IronPython's Zero-Based Arrays*

```
def MyName():
     firstName = raw_input("Please enter your first name: ")
     lastName = raw_input("Please enter your last name: ")

     print "The first character in your first name is", firstName[0]
     print "The first character in your last name is", lastName[0]

MyName()
```

```
Please enter your first name: Alan
Please enter your last name: Harris
The first character in your first name is A
The first character in your last name is H
```

You can also modify the previous code quite easily to extract ranges of characters from a given string. This can be of particular use when handling form inputs. For example, database fields you plan to fill with user input have a given length that you may need to enforce. Extracting only the appropriate number of characters ensures that you are able to fill that field correctly.

Listing 3-5. *Extracting a Range of Characters from a String Input*

```
def MyName():
     firstName = raw_input("Please enter your first name: ")
     lastName = raw_input("Please enter your last name: ")

     print "The first 3 characters in your first name are", firstName[0:3]

MyName()
```

```
Please enter your first name: Alan
Please enter your last name: Harris
The first 3 characters in your first name are Ala
```

Before we move on to numerical operations, let's look at one more particularly useful string operation: the built-in **len** function. The *len* function returns an integer value that equals the total number of characters in a string (Listing 3-6).

Listing 3-6. *Determining the Length of a String*

```
def MyName():
     firstName = raw_input("Please enter your first name: ")
     lastName = raw_input("Please enter your last name: ")

     print "Your name is", len(firstName) + len(lastName), "characters long."

MyName()
```

```
Please enter your first name: Alan
Please enter your last name: Harris
Your name is 10 characters long.
```

A Quick Software Development Detour

Up to this point, all of our code has involved a single method. That is fine for small examples, but we're going to change things a bit moving forward now that we have established some comfort with IronPython. We're going to separate our code further into granular methods. This will make development and maintenance much easier down the road.

If you look at Listing 3-6 again, you'll see that we are violating a few software design principles there. The one I want to draw attention to first is **separation of concerns**. Specifically we have a catch-all method called *MyName* that performs multiple operations: it gets user input in addition to displaying the output. As developers we have a responsibility to write clean code that is easy to maintain. One way to accomplish this is to make sure that each method in our code is responsible for only one task at a time. Let's take the time right now to modify that program (Listing 3-7), and we'll continue this line of thinking as the book continues.

Listing 3-7. *Separation of Code into Methods*

```
def GetFirstName():
     firstName = raw_input("Please enter your first name: ")
     return firstName

def GetLastName():
     lastName = raw_input("Please enter your last name: ")
     return lastName

def DisplayResult(firstName, lastName):
     print "Your name is", len(firstName) + len(lastName), "characters long."

DisplayResult(GetFirstName(), GetLastName())
```

```
Please enter your first name: Alan
Please enter your last name: Harris
Your name is 10 characters long.
```

Now we have a method that gets the first name from the user, a method that gets the last name from the user, and a method that displays the desired results back to the user. We have begun using the **return** keyword to send values back to the calling method, and this is important to understand. If we call a method and need to extract a value from it,

we use the return keyword to tell the method what values to return. Thereby, when one method calls a second method, the second method can return a value back to the first method. This is how we allow each method to have its own distinct task to complete without necessarily **coupling** it to other methods in the program.

■ **Note** *Coupling* and *cohesion* are two software development terms you will frequently hear together in the same sentence, usually phrased something like “loose coupling, high cohesion.” *Coupling* refers to the degree a method in your code relies on other methods to do its job. You want methods to be loosely coupled. My code to display content to the screen should not rely on the input method; it should be concerned only with the task of displaying content. In fact, the code in Listing 3-7 could stand to have the code that computes the length of the first name and last name variables decoupled and placed in their own methods. *Cohesion* refers to the degree that the methods in a particular area of code are related to one another in terms of purpose and functionality. You don’t want a lot of code related to, say, file operations in the same class as your user login code. This makes it hard to understand what a unit of code is trying to accomplish. Loosely coupled, highly cohesive!

The *DisplayResult* method now has two **parameters**, which happen to be the *GetFirstName* and *GetLastName* methods. When the IronPython interpreter hits the DisplayResult line, it reads the two parameters and goes, “I need to execute these two methods and return any values within them.” Those values get passed along to the *DisplayResult* method.

Back on Track

Now that we have a little experience with separation of concerns, we can begin to create more complex IronPython applications. We will create a more complex application using our new skills and exploring the other data types available to us. For our complex application, we’re going to make a word-finding tool. We will provide a word to the program that we want to flag; then we’ll provide a block of text that we want to search. If the word is in that text, the program will tell us at what numerical position it was found (Listing 3-8).

Listing 3-8. *Finding a Word in a Block of Text*

```
def GetWord():
     word = raw_input("Please enter the word you want to find: ")
     return word
def GetText():
     text = raw_input("Please enter the text block: ")
     return text
```

```
def DisplayResult(word, text):
     position = text.find(word)
     if position > -1:
          print "The word", word, "was found at position", position, "."
     else:
         print "The word", word, "was not found in this string."

DisplayResult(GetWord(), GetText())
```

```
Please enter the word you want to find: dog
Please enter the text block: The quick brown fox jumps over the lazy dog.
The word dog was found at position 40 .
```

We accomplished the word-finding task by using the built-in method **find**. This is a very useful method for determining the presence of a piece of data in another piece of data or group of data points. The *find* method is case-sensitive; if you try to find "D" in the preceding example, you won't get any results.

■ **Note** Bearing in mind that IronPython is built around CPython, we recognize that it shares those underlying characteristics. Arrays are always zero-based, meaning that the first value in an array is always at position 0.

IronPython fully supports the concept of **dictionaries**. Dictionaries, in the programming sense, are composed of **key-value** pairs that allow a developer to look up quickly a value for a given piece of data, much like a real-world dictionary (Listing 3-9). Where an array like the types we've seen before use integers (starting with 0) to index elements in the set, a dictionary is indexed by its keys and can use any **immutable** data type for a key. Keep your keys unique; keep your values however you like!

■ **Note** Key-value pairs are very important concepts in software development. You can find them all over; they are used in configuration files, the querystring values in a web page URI (Uniform Resource Identifier), and so on. Different languages support them in different ways, including generics and hash tables.

Listing 3-9. *Using a Dictionary to Store Employees*

```
def CreateDictionary():
     dict = {"Jane":"CEO", "Tom":"CIO", "Susie":"VP", "Bob":"VP"}
     return dict

def GetName():
     name = raw_input("Please enter an employee\'s name: ")
     return name

def DisplayResult(dict, name):
     if dict.has_key(name):
          print name, "is a", dict[name], "in this company."
     else:
          print name, "does not work for this company."

DisplayResult(CreateDictionary(), GetName())
```

```
Please enter an employee's name: Susie
Tom is a VP in this company.
```

Now we have the ability to find items in collections, be they single words in a string or items in a list. Being able quickly to index, sort, and parse data is often critical in any given application. Later, when we look at web development, we will build a rudimentary web crawler, and dictionaries will quickly become the backbone for the entire application. Look out, Google!

Floating-Point Numbers

We addressed integer operations in Chapter 2. You may recall that the integer value is a whole number expressed without fractional or decimal notations. In contrast, floating-point numbers are expressed using fractional or decimal notation. Where a group of integers looks like {1, 2, 3, 4}, the same numbers expressed in floating-point form would be {1.0, 2.0, 3.0, 4.0}. The word *floating* is applied because the decimal point position is not fixed. A set of floating-point numbers might look like {131.0005, 9.154, 2.511, 3.14} and can have a variable number of digits. Fixed-point numbers are a subset of floating-point numbers. Fixed-point numbers can have only a specific number of digits to the right of the most significant digits. So a set of fixed-point numbers at five-digit precision might look like {8000.1, 4302.2, 5115.3, 1098.3}.

Note Floating-point numbers are an absolutely critical aspect of software development. They show up in fuzzy logic and neural network programming and in all manner of mathematical software. If math isn't your strong suit, don't worry. We're not going to be doing anything more difficult than addition, subtraction, multiplication, and division, and IronPython is great at doing all those operations.

IronPython handles big numbers—very big (Listing 3-10). It's quite adept at working quickly with these numbers, and that's important because in computationally heavy systems every bit of performance counts. Stability in dealing with large numbers can also be critical; your application isn't of much use if it goes up in flames every time it gets a number that's pretty large! This is where dynamic typing comes in handy. IronPython will make your variable a floating point as needed, thereby preventing many of the arithmetic overflow problems that can plague statically typed languages.

Listing 3-10. *Big numbers!*

```
def AddBigIntegers():
     bigInteger = 123456789098712390847 + 23456789109182743087➥
+ 345678912091873 + 567891230987230987
     return bigInteger

def AddBigFloats():
     bigFloat = 90871234.12341324 + 8917623.987124➥
+ 1987345.987247 + 43793873.9871234
     return bigFloat

def DisplayResult(bigInteger, bigFloat):
     print bigInteger
     print bigFloat

DisplayResult(AddBigIntegers(), AddBigFloats())
```

```
147481815117794456794
145570078.085
```

Booleans

Is the following statement true or false? *Boolean values are pure logic.* How much simpler can you get than true or false? That's exactly what a Boolean value is, true or false. It does sound very simple, but it's an extraordinarily critical notion. At the most basic level, the software running on a computer system is nothing more than a series of ones (true) and zeroes (false). There's a lot of power in the Boolean values. A typical use of Boolean values is checking whether or not a specific flag is enabled (for example, is there gasoline in your car or not?). See Listing 3-11.

Listing 3-11. *Using a Boolean Value for a Flag*

```
def SetFlag():
     flagEnabled = True
     return flagEnabled

def DisplayFlag(flagEnabled):
     if flagEnabled == True:
          print "The flag is set to true."
     else:
          print "The flag is set to false."

DisplayFlag(SetFlag())
```

```
The flag is set to true.
```

Classes and OOP

If what we took earlier was a software development detour, this next part is the major construction on the highway. The difference here is that this will make your life easier immediately! At our last detour we introduced the notion of separating code into different methods and modules so that there would be a proper division of concerns in our code and to make development and future development easier to manage. We're going to take that concept a step further and separate our code into **classes**, because we're about to get knee deep into **object-oriented programming (OOP)**.

Object-oriented programming is a method of writing software where the emphasis of design is on creating **objects**, which are essentially the virtual representations of either

abstract or tangible items. These objects may contain attributes, behaviors, data, or any combination of these. Certain design principles must be adhered to for a developer writing OOP code, so let's break these principles down.

- *Inheritance*: Objects can inherit traits from other objects, which essentially creates a parent–child relationship between objects. For example, a CEO can inherit the traits of an employee, who inherits the traits of a human being. The human being object is the generic parent object, and employees and CEOs are subobjects that are increasingly specific and customized.
- *Encapsulation*: Objects can wrap up behaviors and data specific to that object, hiding the implementation details from other objects. For example, a CEO needs to have code specific to the particular job of being a CEO, but the *human being* class doesn't need to know that information. The specifics of being a CEO would be encapsulated in the *CEO* object because that is where they are most relevant.
- *Polymorphism*: An object can have a method that performs the same operation on different data types. For example, you may need to create code that examines lists of data. For a fictional business it may be lists of employees, salaries, days off, or any other type of data. Your *business* object could use one method to process those lists, without needing a separate method for every type of list under the sun. This can be achieved by overriding methods, inheriting from other classes, and so on. See Chapter 6 for specific examples of polymorphism as well as how to apply these principles to interacting with a statically typed language.

We are not going to spend a lot of time on theory; we're going to move right into code. Time is money! For this example we will work through the code step-by-step to see how and why we are applying object-oriented principles. This is going to be a big exercise, our first big leap into developing real software with IronPython. The end result will actually be a bit simplistic, but this is a play-by-play of writing classes in an object-oriented fashion and is essential for you to learn as a developer.

CLASSES AHOY!

For this exercise, we're going to implement the *human being*, *employee*, and *CEO* classes described earlier, in the OOP discussion. This is going to be screenshot-heavy and highly detailed because a lot is going on and there's much to discuss as we go, so please excuse any hand-holding!

1. Open up the text browser of your choosing and create a file called *humanBeing.py*.
2. Type the code in Listing 3-12 exactly as it appears into *humanBeing.py* and save the file.

Listing 3-12. *humanBeing.py Implementation*

```
class Human:
   # some data about our human, with default values for convenience
   age = 25
   name = 'Alan Harris'
   weight = 190

   # methods for setting attributes about our human
   def setAge(self, age):
        self.age = age
   def setName(self, name):
        self.name = name
   def setWeight(self, weight):
        self.weight = weight

   # methods for getting attributes about our human
   def getAge(self):
        return self.age
   def getName(self):
        return self.name
   def getWeight(self):
        return self.weight
   def getInfo(self):
        print "Human being", self.name, "is", self.age, "years old➥
and weighs", self.weight, "pounds."
```

■ **Note** What's this "self" business we see scattered about Listing 3-12? Simply put, *self* is how an instance of the object you're using can refer to its own data and methods. When we cover instantiation a few steps from now, you'll see what I mean more clearly.

---

3. Type **ipy c:\python\humanBeing.py** and press Enter. You should see a result like the following screenshot, which indicates that the interpreter has completed successfully. If the interpreter finds errors in your code, verify that everything matches Listing 3-12 and perform this step again. Be aware that this does not ensure that your code works properly! This only helps catch syntax errors at this initial step.



4. Type **ipy** and press Enter. This should open the IronPython interpreter.

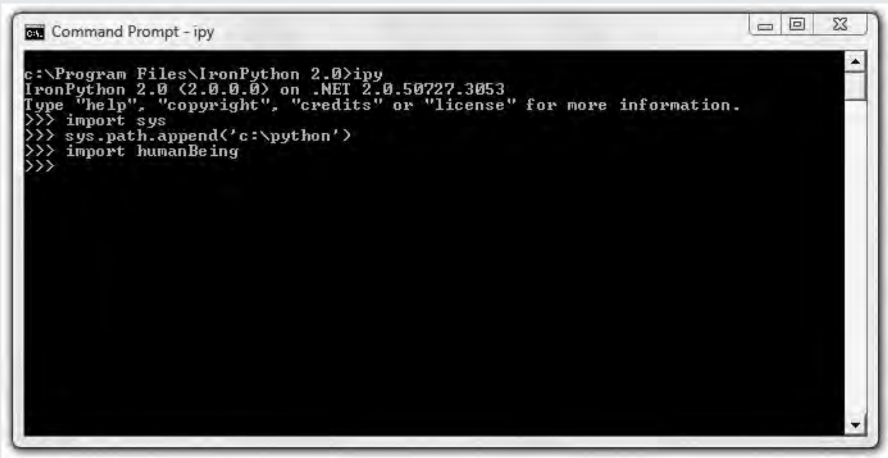
5. At the >>> prompt, type **import sys** and press Enter. You should immediately be presented with another prompt (as in the next screenshot). The *import* command serves to make additional code and functionality available for use. IronPython has a lot of features, and we don't always need them all. In the next steps, you'll see how to use it to provide access to the code you created for the *Human* class.



6. Now type **sys.path.append('c:\python')** and press Enter. This is the path I recommended earlier in the book for storing your IronPython code. If you have placed your code elsewhere, substitute the correct location. You should immediately see another prompt.

Note What have we just done? The step just performed tells the IronPython interpreter that we would like to include the location C:\Python as a valid place from which to import modules. If we didn't do that, the interpreter wouldn't know we had code in that location and we'd get an error in the next step.

7. Type **import humanBeing** and press Enter. What this tells the IronPython interpreter is that you want to add all the code in a file called *humanBeing.py* to your current environment so that you can use it. The *.py* extension is assumed, so you don't need to add it. After you type this and press Enter, your screen should look like the following screenshot.



8. Having successfully imported the *humanBeing.py* file to the environment, let's take it for a spin and see what it can do for us. The file name *humanBeing* is used to set up automatically a **namespace** for our code to live in, so we will need to employ that namespace as a prefix when working with classes we've created.

Note A *namespace* is nothing more than a grouping that allows you to keep naming collisions down to a minimum. Say you created a handful of classes that have names identical to someone else's class names. If you were to try to reference one of yours in your code, the interpreter would have no way of knowing which one you wanted! Computers are powerful but dumb; they need your explicit instruction to realize what you're trying to accomplish. Providing a namespace lets you prefix your classes with a meaningful name that allows the interpreter to say, "Oh, I get it, you want *this* particular thing to happen."

9. Now that we have told the interpreter where to find our class and successfully imported it, let's try it out.

```
>>> neighbor = humanBeing.Human()
```

10. When we have a class, it does not yet refer to any particular object. We have a class called *Human*, but which human are we talking about? So for this example we made a *neighbor* object and **instantiated** it as a Human, meaning the *neighbor* object is now one instance of the *Human* class. We can instantiate as many Humans as we like, and we'll do just that shortly, but first we should try to use one of the methods that a *Human* class provides.

■ **Note** Remember how I said you'd see the *self* parameter used more clearly? If you look at what we just did and at the underlying *Human()* method of the *humanBeing* class in your *humanBeing.py* file, you will see what the use of *self* just allowed. We instantiated a class of *Human* as an object called *neighbor*, and *neighbor* is passed into the methods to fulfill the *self* parameter and to tell the interpreter which Human we're dealing with. You don't actually have to use the word *self*; you could call it whatever you like. I use *self* because it's convenient and because I am aware that many other developers and authors use it, so I would like to maintain some convention as you read other people's code.

```
>>> neighbor.getInfo()
Human being Alan Harris is 25 years old and weighs 190 pounds.
```

11. The *Human* class has a method called *getInfo* that displays information about the particular instance of *Human* that we have. In this case, we have not provided any details that override the default values we provided, so it returns those defaults. Methods are called in the format *object.methodName*.

12. Now let's use our *Human* class to make another instance of a Human, one with different traits.

```
>>> friend = humanBeing.Human()
>>> friend.setAge(55)
>>> friend.setName('Tom Smith')
>>> friend.setWeight(160)
```

13. We've created two Humans now, one called *neighbor* and one called *friend*. They are both Humans, but they are specific instances of Humans with unique data in them. Let's prove that point.

```
>>> neighbor.getInfo()
Human being Alan Harris is 25 years old and weighs 190 pounds.
>>> friend.getInfo()
Human being Tom Smith is 55 years old and weighs 160 pounds.
```

14. Now, that's pretty cool. If you've been following along on your computer, your screen should look something like the following.



15. Let's make things more specific. We are going to modify our code to make a class called *Employee* that will be more specific than a Human. *Employee* will inherit traits from *Human* (since this clearly doesn't work in reverse!). By creating an Employee, we will be creating an *Employee* instance that is by default also an instance of a Human. Quit the interpreter by typing **exit()** and pressing Enter. Open your *humanBeing.py* file and add to the bottom of the file the code in Listing 3-13; then save it.

Listing 3-13. *humanBeing.py Continued*

```
class Employee(Human):
     # some basic data about our employee
     payrate = 10
     hours = 40

     # methods to set that data
     def setPayRate(self, payrate):
          self.payrate = payrate
     def setHours(self, hours):
          self.hours = hours
```

```
    # methods to retrieve that data
    def getPayRate(self):
        return self.payrate
    def getHours(self):
        return self.hours
    def getEmployeeInfo(self):
        print "Current pay rate is", self.payrate, "dollars per➥
hour at", self.hours," hours per week, which totals $",➥
 self.hours * self.payrate, " weekly."
```

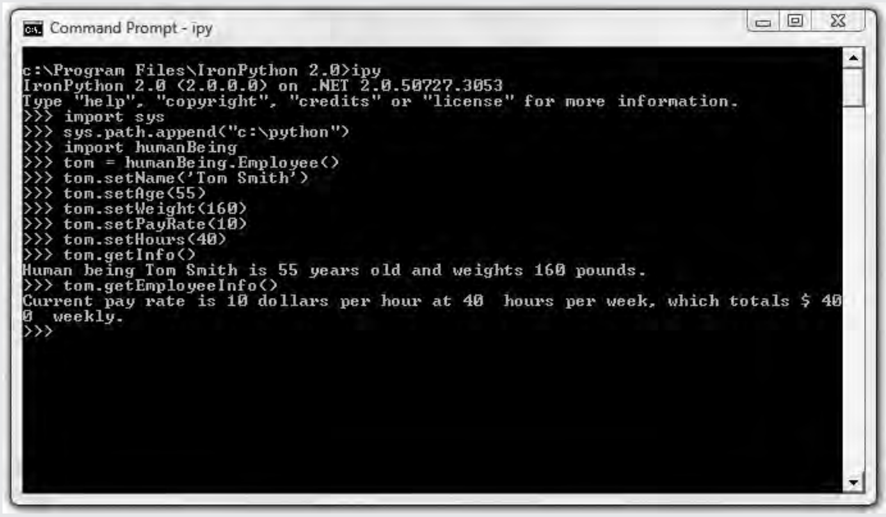
16. Now we have an *Employee* class that inherits traits from the *Human* class. Note that in the definition for *Employee* we have `(Human)` at the end. This is how inheritance is specified; we're telling the interpreter that *Employee* inherits from *Human*. In a moment we'll prove that point. Similar to the *Human* class, we've got a few data values and ways to display them. So let's create some employees and try this out. Open the interpreter again, import *sys*, add your Python directory, and import the *humanBeing* file.

■ **Note** IronPython, like CPython, supports multiple inheritance. That is, a class can inherit from more than one base class. You will find this is not the case in other .NET languages, such as VB .NET and C#; they support only single inheritance. Multiple inheritances can be a maintenance nightmare! It's an oft-criticized feature of certain languages, and you won't see me making use of it in this book.

Technically speaking, although other .NET languages tend to support only single inheritance, you can generally implement multiple interfaces. Interfaces define a contract, specifying the methods and properties that a class must completely implement to meet the criteria of the interface.

```
>>>  tom = humanBeing.Employee()
>>>  tom.setName('Tom Smith')
>>>  tom.setAge(55)
>>>  tom.setWeight(160)
>>>  tom.setPayRate(10)
>>>  tom.setHours(40)
>>>  tom.getInfo()
Human being Tom Smith is 55 years old and weighs 160 pounds.
>>>  tom.getEmployeeInfo()
Current pay rate is 10 dollars per hour at 40 hours per week,➥
which totals $ 400 weekly.
```

17. If all went well, you should see something like the following screenshot. Our *Employee* class is derived from our *Human* class. Whereas the behavior specific to an Employee is hidden from a Human, an Employee needs to know how to be a Human. This is an example of an **is a** relationship, in object-oriented terms. An Employee "is a" Human, but a Human is not necessarily an Employee.



■ Note Relationships in object-oriented terms are generally broken down into "is a" and "has a" varieties. For example, a dog "is an" animal and "has a" heart. "Has a" relationships describe traits that an object possesses, and "is a" relationships describe what an object is. We'll see more of this as the book progresses. In our example here, Tom "is an" Employee, which "is a" Human, and he "has a" pay rate of 10 dollars an hour. It's really nothing more complex than that.

---

18. We're almost done! All we need to complete our exercise is to add the *CEO* class to this code and then to make use of it. Close the interpreter and open *humanBeing.py* again, add to the bottom of the file the code in Listing 3-14, and then save it.

Listing 3-14. *humanBeing.py Continued*

```
class CEO(Employee):
     # some data about the CEO
     bonus = 5000
     annualGrowth = 2.3

     # methods to set that data
     def setBonus(self, bonus):
          self.bonus = bonus
     def setAnnualGrowth(self, annualGrowth):
          self.annualGrowth = annualGrowth
     # methods to get that data
     def getCEOInfo(self):
          print "The CEO's end-of-year bonus is $", self.bonus,➥
 " * ", self.annualGrowth, "% annual growth, totaling $",➥
self.bonus * self.annualGrowth, "."
```

19. Open the interpreter again, import everything you need from the preceding, and try the following code.

```
>>>  tom = humanBeing.CEO()
>>>  tom.setName('Tom Smith')
>>>  tom.setAge(55)
>>>  tom.setWeight(160)
>>>  tom.setPayRate(70)
>>>  tom.setHours(40)
>>>  tom.setBonus(5000)
>>>  tom.setAnnualGrowth(2.3)
>>>  tom.getInfo()
Human being Tom Smith is 55 years old and weighs 160 pounds.
>>>  tom.getEmployeeInfo()
Current pay rate is 70 dollars per hour at 40 hours per week,➥
which totals $ 2800 weekly.
>>>  tom.getCEOInfo()
The CEO's end-of-year bonus is $ 5000 * 2.3 % annual growth, totaling
$11500.0 .
```

20. Type **exit()** and press Enter to exit the interpreter. You're done! The entire listing of *humanBeing.py* is provided on the following pages.

Listing 3-15. *Complete Listing of humanBeing.py*

```
class Human:
     # some data about our human, with default values for convenience
     age = 25
     name = 'Alan Harris'
     weight = 190

     # methods for setting attributes about our human
     def setAge(self, age):
          self.age = age
     def setName(self, name):
           self.name = name
     def setWeight(self, weight):
          self.weight = weight

     # methods for getting attributes about our human
     def getAge(self):
          return self.age
     def getName(self):
          return self.name
     def getWeight(self):
          return self.weight
     def getInfo(self):
          print "Human being", self.name, "is", self.age,➥
"years old and weighs", self.weight, "pounds."

class Employee(Human):
     # some basic data about our employee
     payrate = 10
     hours = 40

     # methods to set that data
     def setPayRate(self, payrate):
          self.payrate = payrate
     def setHours(self, hours):
          self.hours = hours
```

```
    # methods to retrieve that data
    def getPayRate(self):
        return self.payrate
    def getHours(self):
        return self.hours
    def getEmployeeInfo(self):
        print "Current pay rate is", self.payrate,➥
"dollars per hour at",➥
self.hours," hours per week, which totals $",➥
self.hours * self.payrate, " weekly."

class CEO(Employee):
    # some data about the CEO
    bonus = 5000
    annualGrowth = 2.3

    # methods to set that data
    def setBonus(self, bonus):
        self.bonus = bonus
    def setAnnualGrowth(self, annualGrowth):
        self.annualGrowth = annualGrowth
    # methods to get that data
    def getCEOInfo(self):
        print "The CEO's end-of-year bonus is $", self.bonus,➥
" * ", self.annualGrowth, "% annual growth, totaling $",➥
self.bonus * self.annualGrowth, "."
```

That's not a bad raise for Tom! Tom has gone from being a mere human, to being an employee, to being a wealthy CEO in a few short pages. By now you should have a pretty good grasp of the fundamentals of classes in IronPython, as well as some understanding of object-oriented principles and how they apply to business solutions. Granted, the human-to-employee-to-CEO example is rather simple when compared to the modeling of enterprise solutions, but it is very effective at demonstrating how an OOP developer thinks when approaching a problem. Breaking problems down into simpler pieces and separating concerns into small, easy-to-maintain components will make you a better developer. And when you return to do maintenance on a piece of code you haven't touched in six months or a year, you'll thank yourself for making things easy at the beginning.

.NET Data Types

In the .NET world, most of what we've covered still rings true. Developers are still working with strings and numbers and so on. The .NET framework takes a different route to the destination, however; regardless of whether you're dealing with a value or reference type, everything in .NET is an **object**.

■ **Note** *Object* is the superclass that all types inherit from in .NET. It doesn't matter if you're working with a primitive integer or some fancy *GetTerrificProgramFactory* class—the foundation on which it's constructed is *object*.

What does it mean to have every type derive from *object*? Essentially it creates a unified type system, allowing any class or data type to be treated as an object. This is a construction of a statically typed language; IronPython by itself does not care for or need this information. When working with other .NET classes and code, it is important to understand this concept and its implications. We can also take advantage of its benefits.

Value and Reference Types

In .NET, a *value type* is a variable whose data is stored directly within the memory assigned to that particular variable. Integers, Booleans, and strings are all examples of value types. Classes are reference types; when a *reference type* is created, space is allocated in memory for an object, and then instances of that object are created. The code in Listing 3-16 shows the implementation differences between the two types.

Listing 3-16. *Value Types vs. Reference Types*

```
# these are value types; they are not instances of objects
speed = 55
driving = true
driver = "Speedy"

# these are reference types; they are instances of objects
vehicle = vehicleTypes.GetCar()
road = roadmaps.GetHighway()
```

Mixing and Matching

The wonderful thing about IronPython development is that if you're a seasoned Python developer, you can use all the same syntax you're comfortable with and simply take advantage of .NET framework classes when you like. If you're a seasoned developer in another .NET language, you can freely take advantage of traditional Python code and constructs to use in your code.

MIXING .NET WITH PYTHON

Open up the IronPython interpreter. For this exercise, we'll create a few types using the traditional Python syntax and perform various operations on them using classes from .NET.

```
>>>  import clr
>>>  clr.AddReference('System')
>>>  from System import *
>>>  foo = "bar"
>>>  if (String.IsNullOrEmpty(foo)): print "Foo has no value."
… (press Enter at this prompt. Nothing should be displayed.)
>>>  if not (String.IsNullOrEmpty(foo)): print "Foo has a value."
… (press Enter at this prompt.)
Foo has a value.
>>>  Int32.Parse("12345")
12345
>>>  number = 7
>>>  print number.ToString()
7
>>>  dummy = "the/quick/brown/fox"
>>>  print dummy.Split('/')
Array[str](('the', 'quick', 'brown', 'fox'))
>>>  print dummy.Split('q')
Array[str](('the/', 'quick/brown/fox'))
>>>  sentence = "The only thing we have to fear is fear itself."
>>>  print sentence.Substring(0, 30)
The only thing we have to fear
```

Summary

We've covered the basic Python data types and how to perform common operations on them. We have learned how to store data in a few different types of structures and how we can retrieve that information in a straightforward fashion. We have covered very large numerical data types in IronPython and dealt with Boolean values. Finally we created a variety of classes to demonstrate object-oriented software development and how IronPython supports development of advanced systems, and we discussed how we can mix and match IronPython types with the .NET framework.

CHAPTER 4

IronPython Studio

"Once a new technology starts rolling, if you're not part of the steamroller, you're part of the road." — Stewart Brand

One of the best resources that programmers have is an **integrated development environment,** or IDE. Up until now, we've been doing all our programming using the IronPython console interpreter, which is a viable way to program but not the easiest or fastest. Luckily for us, Microsoft has an IDE solution integrated with Visual Studio 2008 that makes developing and managing IronPython applications downright *enjoyable*. This IDE, called IronPython Studio, is a free download from Microsoft that facilitates development of IronPython **console** and **Forms applications**. Before we get started installing and working with the IDE, let's take a look at where we're going regarding application development and why an IDE is such a useful tool in a programmer's arsenal.

Hopping Onto the Steamroller

So far, you should be pretty well versed in building a console application; that's all we've really done so far. Such applications are a terrific way to begin learning a programming language because you don't have to concern yourself with the shiny veneer of a typical Windows application and the programming tasks that come along with developing software in that type of environment. Console applications are great for trying out ideas or algorithms quickly, and they make wonderful **test beds** for code libraries and snippets you will write throughout your programming career.

Note A *test bed* is a platform or environment where you can test software in a repeatable way. An example from my real-life development is an ASP .NET project I created at work called *Sandbox*. The Sandbox project is set up in such a way that I can reference code I've written elsewhere and measure performance, perform automated testing to make sure I'm getting the results I expect, and so on. For smaller situations, I'll frequently make a console application and run my code in it to see how things are working. We'll cover these types of development situations in detail a bit later and work through some examples firsthand.

Forms applications are the typical style of software you're probably used to working with in a Windows environment. They generally use the standard Windows menus and controls to maintain a consistent look and feel when possible, although you can get very creative with the interface (sometimes *too* creative!). In the context of Windows, they use what is called an **event-driven** programming model, meaning that the application is capable of responding to and acting on a variety of events that could be initiated by the operating system, the user, or another program entirely.

An event-driven programming model is a bit different than the type of programming we've done so far in this book, although there are a few similarities. Responding to events requires that the program operate in a continual loop, checking for inputs, processing them if available, and then updating the user interface so that the end user knows things are happening. In a very limited sense, we have done that ourselves in previous examples. The problem is that in those examples we did not implement a true event-driven system. We have been generally operating in a very linear, one-way style: the program asks the user for input and quietly twiddles its thumbs until that input is provided. Assuming that the system always has power and never crashes, if a user never provides that input, the system will be stuck in that state indefinitely, unable to perform any other tasks until it has gotten past that one particular roadblock. Our poor programs are waiting at a perpetual red light with their blinkers on.

We're going to put all of these things in perspective with some IronPython code that builds a basic Forms application by hand. We'll see what happens when an event loop is set up and why our lives will be so much easier throughout the rest of the book, because we'll be using IronPython Studio to handle some of the heavy lifting. With the text editor of your choice (for me, it's Notepad—I feel so low-tech sometimes), enter the code from Listing 4-1 exactly as it appears and save it as *form.py*.

Listing 4-1. *An Implementation of a Forms Application*

```
import sys
import clr

clr.AddReference("System.Windows.Forms")

from System.Windows.Forms import Application, Form

class IronPythonForm(Form):

    def __init__(self):
        self.Text = 'IronPython Forms Application'
        self.Name = 'FormApp'

form = IronPythonForm()
Application.Run(form)
```

Whoa! This whole IronPython thing just got *real*. In the immortal words of Douglas Adams, "Don't panic." Go ahead and run this script. You should see something along the lines of what's displayed in Figure 4-1.

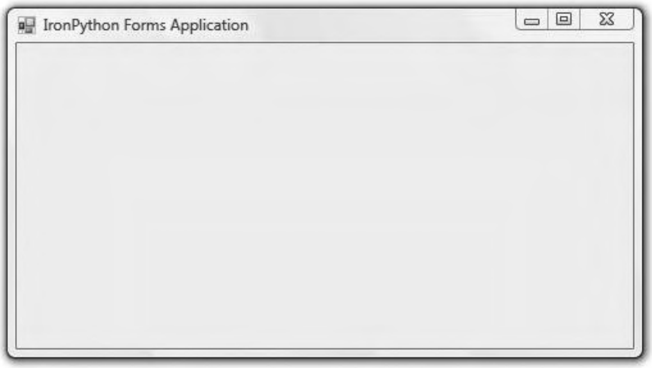


Figure 4-1. *It's not terribly interesting, but it does run!*

You have to admit, that's pretty cool for so little effort. Now, some of the code syntax is going to look familiar, and some of it looks just plain cryptic. Let's go line by line and really examine what's happening here, for this is the foundation on which our applications will be built.

Import sys and *import clr* allow us to use functionality that is contained in other modules or **assemblies**. We've used this command before, but not in terms of the CLR assembly. Importing the CLR assembly lets us tap into the vast functionality of the .NET framework and is really the starting point for the rest of this book. The CLR assembly exposes a large number of namespaces and methods that let you write code to target the .NET framework as well as saving you the pain (or pleasure) of implementing a lot of this functionality from scratch by yourself.

■ **Note** In .NET lingo, an *assembly* is the smallest unit of code available for deployment. It is a versioned file and can be made up of one or more code files. Basically, it's the building block of .NET software development.

The line `clr.AddReference("System.Windows.Forms")` is where the rubber meets the road, as it were. What we're telling IronPython is: In that *clr* assembly we imported a moment ago, I want to be able to access the methods available to me in the *System.Windows.Forms* namespace. Immediately afterward we told IronPython `from System.Windows.Forms import Application, Form`, which indicates to the compiler some specifics about the names we want to use in our code that can be found in *System.*

Windows.Forms. Then we created a class, `class IronPythonForm(Form)`, which takes a Form object as its parameter.

A form exposes a variety of properties and methods to the outside world. By changing the properties *self.Text* and *self.Name*, we modified the text that appears in the title bar of the application as well as the name of the form.

Next we created an instance of the form by writing `form = IronPythonForm()`, and finally we got everything moving with `Application.Run(form)`, which set up the event loop for us. That's not too bad for 10 lines of code! Although the program isn't terribly useful at the moment, it does exhibit some very important traits: it is a true Windows application, it was created by hand in a dynamic language, and it has a basic event loop for us to use.

You'll notice that although the program isn't doing much, it's also not really sitting around waiting for us to do something either. You can minimize or maximize it, resize it, close it, or leave it sitting quietly while you go about your business. It doesn't care. Underneath the hood, the event loop is running, waiting for incoming messages and input. We'll provide that sort of useful information to it in a bit. Right now we're going to teach you a shortcut.

So Much Typing...Is There a Better Way?

Knowing what's happening at a low level is very, very useful in programming. The more you understand the system with which you're working, the better use you can make of it. However, I don't imagine you want to type that code every time you create a form; it's boilerplate, and we should be able to automate the whole routine. Let's take a moment to install IronPython Studio and see what benefits we can make use of. I'm going to assume that you have installed Visual Studio; it's one of the prerequisites I mentioned in Chapter 1. If you haven't, flip back and do so.

Now you'll need to proceed with the installation of IronPython Studio.

DOWNLOADING AND INSTALLING IRONPYTHON STUDIO

IronPython Studio is available as a free download from the CodePlex web site, and free is always the right price when it comes to a piece of software you want. As I write this chapter, the current version is 1.0.

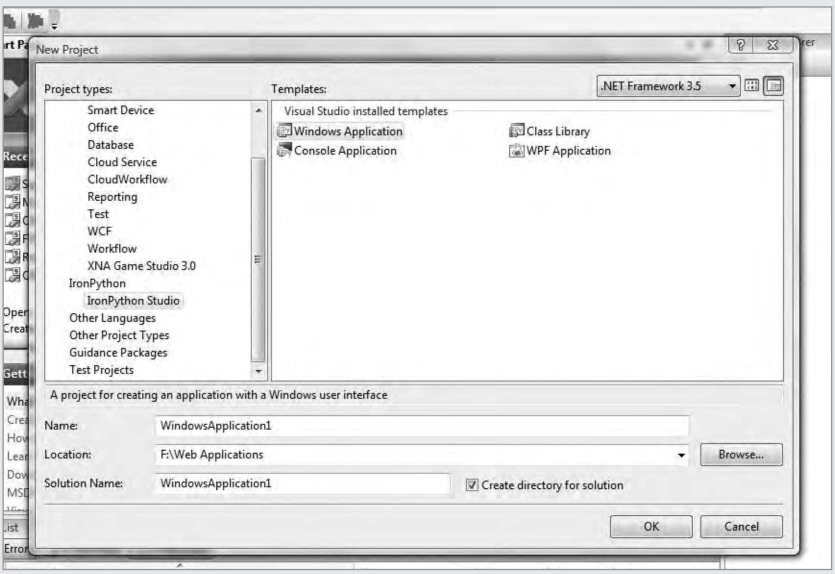
1. In the web browser of your choosing, go to `http://www.codeplex.com/IronPythonStudio`.



2. Click the *IronPythonStudio 1.0 release* link; then on the next screen click the *IronPythonStudioIntegratedSetup* link.

3. Download the zip file, open it, and run the *IronPythonStudioIntegrated* file to install IronPython Studio.

4. Once the installation is complete, open your copy of Visual Studio and select New and then Project. You should see something similar to the image that follows.



The second illustration in the sidebar shows the default project types that IronPython Studio provides. Click the Windows Application template; then, for a Name type, select *form2* and click OK. After the program chugs and works for a few moments, you'll see something like Figure 4-2.

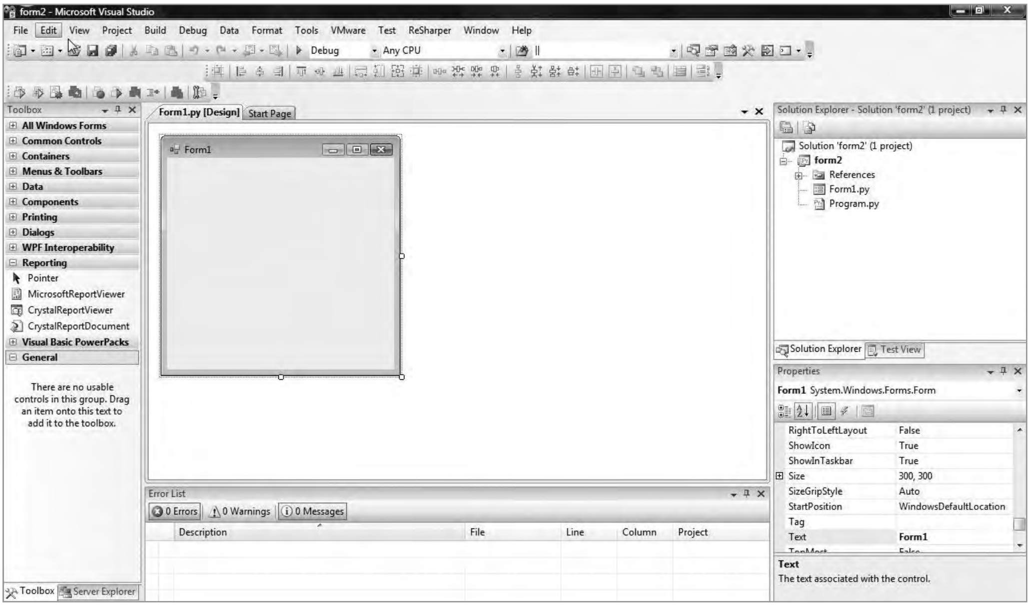


Figure 4-2. *Looks pretty familiar, no? The IDE can reduce your development times significantly.*

Here we see the first major benefit of using the IronPython Studio IDE: *the environment does some of the work for you*. Again, for the sake of learning or understanding, it is *always* a good idea to experiment and try things the manual way where possible. But it's not a good use of your time continually to reinvent the wheel. Letting the IDE handle some of the setup tasks means less code for you to write by hand.

That doesn't mean that IronPython Studio generated the exact same code we used earlier, however. On the right-hand side of the screen, in the Solution Explorer window, right-click on *Form1.py* and select View Code (Figure 4-3).

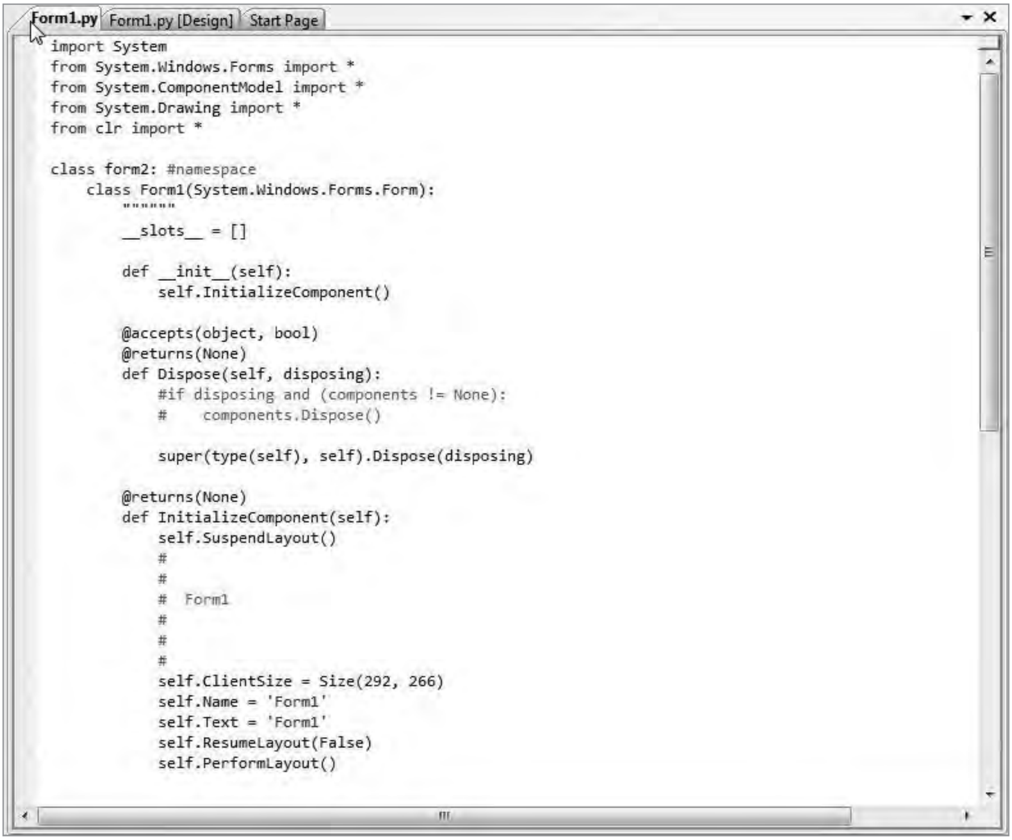


Figure 4-3. *On second thought, maybe it's not entirely familiar.*

Although the code doesn't look quite the same as what we wrote, some of its elements are recognizable. Modify the line `self.Name = 'Form1'` to `self.Name='FormApp'`. Next, modify the line `self.Text = 'Form1'` to `self.Text = 'IronPython Forms Application'`.

■ **Note** Always save your work. It's just a good practice to get into and has the additional benefit of stopping the IDE from complaining to you every time you run the application.

Go ahead and press F5 to build and run this program, or select Start Debugging from the Debug drop-down menu. Compare the way the program looks in Figure 4-4 to the program in Figure 4-1.



Figure 4-4. *The end result looks the same.*

They say all roads lead to Rome, and in IronPython development there is an element of truth to that. You can take a variety of ways to get to the same destination. You can implement a lot of features or just the bare minimum to achieve functionality. It depends on the situation and the requirements of the application.

■**Note** Anyone out there who's a fan of "Office Space" should know what happens to people who just do the bare minimum.

Forms, from the Ground Up

It's all well and good to have a basic form on the screen, but we're developers. We demand *action.* It's not good enough to sit there and stare back at me; the application needs to *do* something. Having touched on the event-driven model in Forms application programming, let's try it for ourselves.

First, select the design view of the application by either clicking the Form1.py [design] tab or right-clicking on *Form1.py* in the Solution Explorer and selecting View Designer. With the designer on screen, you will see small white squares on the bottom-right corner of the form, halfway down the right-hand side of the form, and halfway across the bottom of the form. Click and hold down the left mouse button on the white square halfway down the right-hand side of the form, and drag it to the right to resize the form. See Figure 4-5 for a rough idea of the size we're looking for. It doesn't have to be precise, just larger than what we started with.

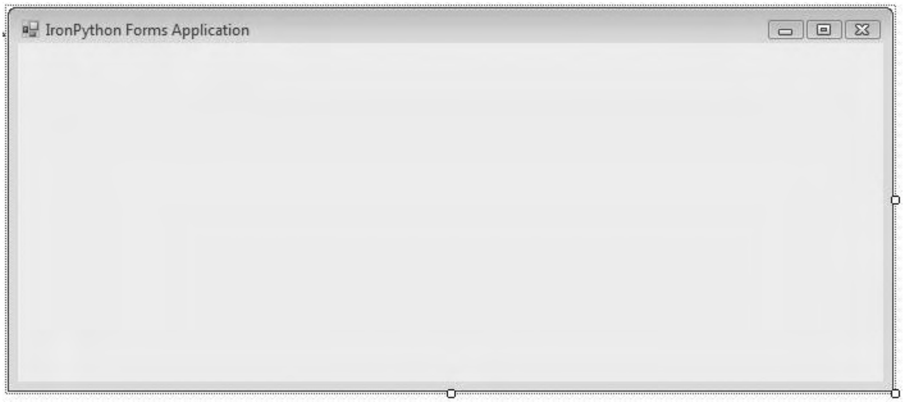


Figure 4-5. *Our form expanded!*

On the left-hand side of the screen, you will see the toolbox, which has expandable tabs that contain objects you can drag and drop onto your form. Many of these elements are standard to .NET, meaning that you can access identical objects if you were building this application in C# or VB.NET. This is a second benefit of the IDE: *common objects and components can easily be added to an application via the toolbox.* For right now, left-click on the Common Controls tab in the toolbox so that it is opened up, and then left-click and drag a Label object and a Button object to the form (Figure 4-6).



Figure 4-6. *We've added two controls to the form.*

■ **Note** One last reminder to save! The compiler will pop up a well-intentioned, if not slightly annoying, message if you haven't saved before trying to run your code.

Once you've got both the label and the button on the form, press F5 to run the application again. Click the button. What happened? Well, a big fat *nothing* happened. But why?

It's All This Substandard Wiring!

Nothing happened because we didn't tell the application what to do if we click the button. Specifically, we didn't **wire** the button to anything. Don't mistake the application's not *doing* anything for the application's not *knowing* we clicked the button. The message was received loud and clear. Unfortunately, that message was "I was clicked, but you don't need to do anything about it."

■ **Note** When a developer *wires* a control, he or she is borrowing a term from the electrical engineering world. A button by itself is useless. If the power switch on your computer weren't wired to anything, it wouldn't *do* anything. You could flip it all day and get no result. The same is true in programming. Until we wire our button to the application by providing some instructions on what to do when it's clicked, it does precisely zilch.

Before we start working heavily with these controls, we really need to give them meaningful names.

■ **Tip** I can't stress how awful a habit it is to leave controls with the default names the IDE provides. When you add a control, the IDE will generally use the name of the control followed by a number that is 1 greater than the previous number of that control on the form. For instance, if you had added five Label controls to the form, the next one would be called Label5. When you're dealing with one or two controls on a form, this isn't too bad. However, once you've got 50 or 100 controls on a form, you'll want to tear your hair out trying to figure out the functional difference between Label39 and Label17. Get into good naming habits early!

Left-click on the Label control on your form. You will see the Properties window on the bottom right of the IDE change to reflect the properties of the object you've clicked. Scroll the list of properties until you see one called (Name), which should be third on the list. Change the name from *Label1* to *lblUpdateText*. Next, left-click on the Button control on your form and scroll the properties list to the (Name) property and change it from *Button1* to *btnUpdate*.

Tip For controls on either a web page or a Forms application, I tend to prefix the control name with a three-letter abbreviation, to let me know what type of control I'm dealing with. This comes in really handy; at a quick glance in my code I can easily identify the buttons, labels, text boxes, or any other control without occupying a lot of space. To me, *LabelUpdateText* is a bit verbose, whereas *lblUpdateText* conveys the meaning just fine. This a personal preference. If you find you don't want or need those prefixes or you prefer some alternative naming convention, by all means do what works best for you. With that said, always keep future maintenance in the back of your mind. That button you made called *PDmc2* to update the sales reports might make total sense now, but I bet a few months from now things might not be so clear.

Now that you've given some more meaningful names to your controls, it's time to do a little wiring. Double-click on the Button control on your form; this should bring up the code window (Listing 4-2). Note that the IDE has inserted a bit of code on your behalf.

Listing 4-2. *A Snippet of the Button-Handling Code the IDE Produced*

```
@accepts(Self(), System.Object, System.EventArgs)
@returns(None)
def _btnUpdate_Click(self, sender, e):
     pass
```

This method signature says that you've defined a method called *_btnUpdate_Click* that accepts *self*, a *sender* object, and a list of event arguments as parameters. The line `pass` can be removed; it's a placeholder in IronPython that does nothing. In place of the `pass` line, add the code shown in Listing 4-3.

Listing 4-3. *Telling the Button to Update the Label Control*

```
     self._lblUpdateText.Text = "The button was clicked, so this text has been➥
updated accordingly."
```

Let's try running the application again; once it's loaded, click the button. You should see the text change from its initial state to the text you just entered (Figure 4-7).



Figure 4-7. *The button was successfully wired and the label updated.*

Clean Code Is Happy Code

Now that we've gotten our button wired up, we should clean up our code a bit. Currently the form has intimate knowledge of the behavior of the button when it's clicked. What we should do is create an IronPython class file and move the code that actually changes the text to it. Right-click on the *form2* project title in the Solution Explorer, select Add, and then click New Item (Figure 4-8).

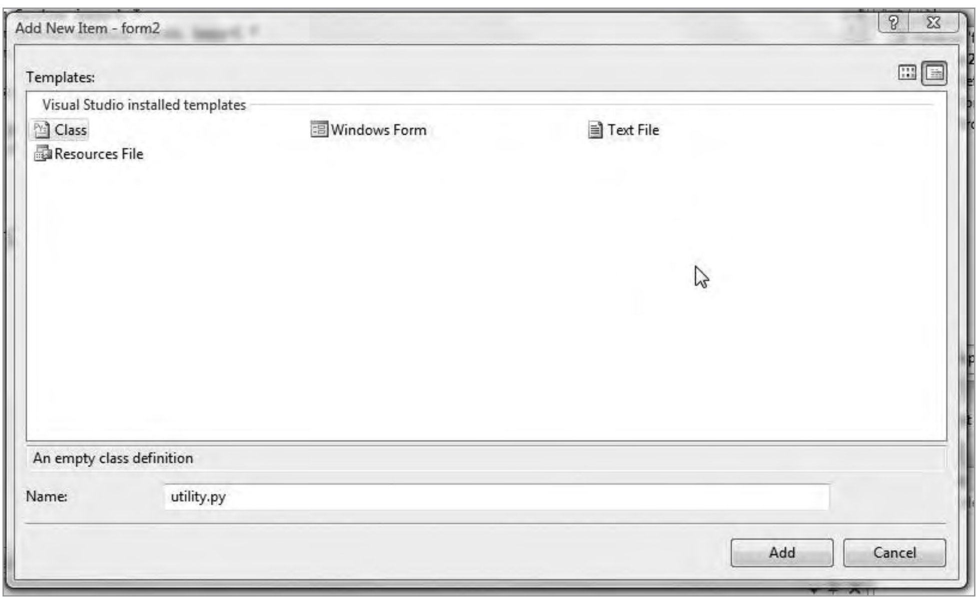


Figure 4-8. *IronPython Studio provides some ready-made code templates as well as project templates.*

Select Class, and name the new class file *utility.py*. When ready, click Add. IronPython Studio will add the file to your current project and automatically open it. It should look like Figure 9-10.

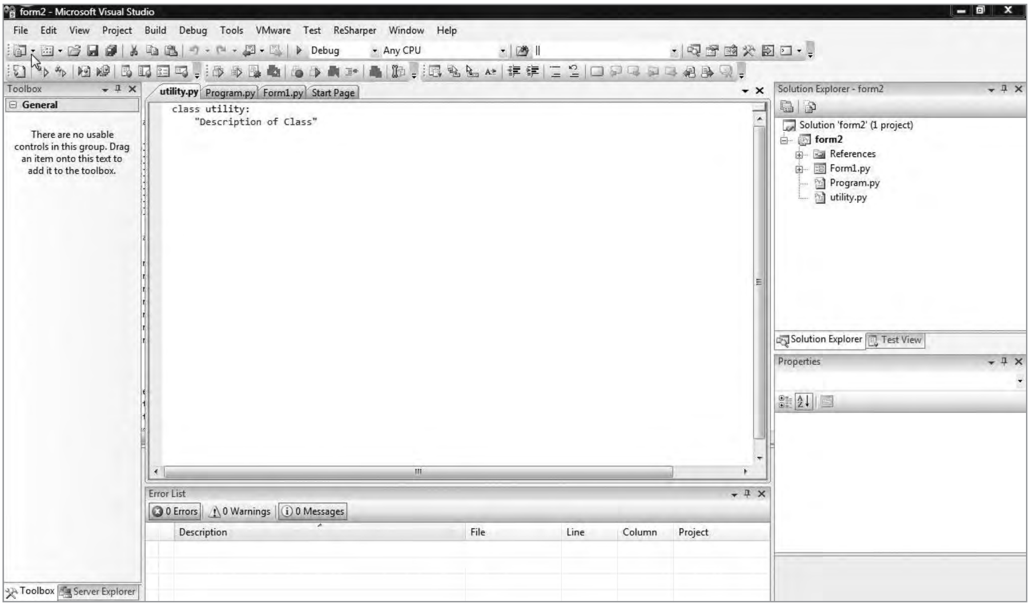


Figure 4-9. *Generated class templates are very sparse initially.*

The desired result is that we will be updating a label with different text than it had initially. Let's go ahead and make a method called *UpdateText* to fulfill this purpose. Enter the code exactly as it appears in Listing 4-4, and save the class file.

Listing 4-4. *Filling in the Functionality of the UpdateText Method*

```
class utility:
    "Some basic IronPython utility methods"

    def UpdateText(self):
        return "The button was clicked, so this text has been updated ➥
accordingly from utility.py."
```

With our *UpdateText* method complete, we should turn our attention back to the button on our form. We need to create an instance of our *utility* class and call the *UpdateText* method when the button has been clicked. Change the code in the *_btnUpdate_Click* method so that it matches what appears in Listing 4-5, and press F5 to run the application.

Listing 4-5. *Updating the Button to Call Our Utility Class*

```
def _btnUpdate_Click(self, sender, e):
     util = utility.utility()
     self._lblUpdateText.Text = util.UpdateText
```

The application should start up fairly quickly and you'll be presented with the same form you've gotten so used to. Click the button. Did the label control update? No? What happened? We were taken back to IronPython Studio where the IDE had flagged an error in my code; see Figure 4-10.

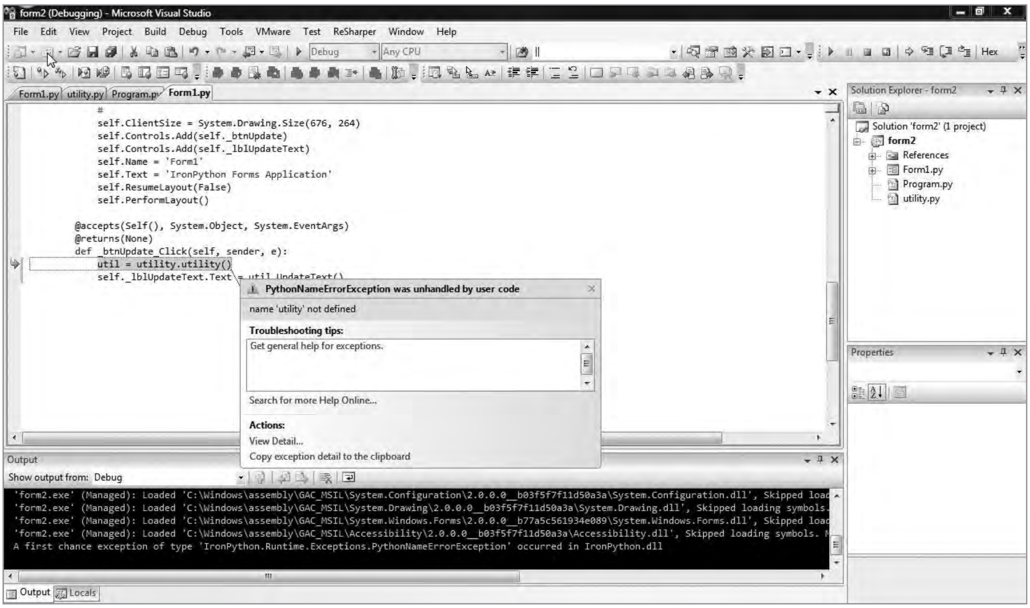


Figure 4-10. *The IDE has pointed us to an error in our code.*

Before we address what this error means and why it's there, we should highlight one of the three major benefits of an IDE over the console interpreter. If we were running this code via the IPY interpreter, then once an error occurred the interpreter would tell us what happened and where, but we would be unable to pause program operation to examine the current state and see precisely what had caused the error. In IronPython Studio, when we run the application and an error occurs, operation pauses and we can get information on the condition of the program, make edits, and so on. Wahoo!

■**Note** You don't only get access to this sort of information when something goes terribly wrong. You can set things called *breakpoints* in your code. When a breakpoint is set on a line of code, program execution pauses and control is returned to IronPython Studio. You can examine variable values using *watches* and do a variety of other debugging tasks to nip problems in the bud. We'll cover these debugging techniques and more in the next chapter.

This is also a terrific demonstration of the nature of a dynamic language versus a static one like C# or VB.NET. The reason the code has failed to run is that we did not import our *utility* module into our form, so the code we wrote to update the text is inaccessible from our current location. In C# or VB.NET, if we attempt to call code that we have not made accessible via the *using* or *Imports* statement, then the IDE will tell us immediately and prevent us from even running the code. In IronPython Studio, the interpreter won't know there's a problem until it hits that line in the script and finds no matching method available to call, and it lets us know about it at runtime instead. It's something to be aware of; in IronPython the IDE is less of a crutch in terms of ensuring that you've done everything correctly before you hit F5 and try it out for yourself.

■**Note** In C#, you can access code in another module or assembly with the *using* keyword. In VB.NET, you would say *Imports*. They both do essentially the same job that *import* does in IronPython.

Let's correct the error in our application and try this again. You can stop the debugging session either by closing the application, by pressing Shift+F5, or by pressing the Stop button on the toolbar. At the top of the code in *Form1.py* you will see numerous import statements above the beginning of the class declaration. We need to add a statement importing our *utility* class for everything to function properly. Modify the code so that it matches what appears in Listing 4-6; then press F5.

Listing 4-6. *Importing the Utility Class with the Relevant Line Boldfaced*

```
import System
from System.Windows.Forms import *
from System.ComponentModel import *
from System.Drawing import *
from clr import *
import utility
     class form2:
```

Tip These namespaces are case sensitive; if you named your class *Utility* instead of *utility*, you'll need to call it accordingly.

As before, the application should build and execute fairly quickly, and you'll see the form pop up on screen. Click the button and see if your result matches Figure 4-11.



Figure 4-11. *Now that we have fixed the error, our utility class is used properly.*

Summary

Now that you've had the introductory tour to IronPython Studio, I'm sure you see the benefits in working with an IDE over working with the console interpreter. It's perfectly fine if you choose to do your work via the command line, but do so knowing that you sacrifice some of the benefits the IDE provides, such as templates and efficient debugging. We've only touched the surface of what IronPython Studio can do for you. As we move forward we'll cover more advanced debugging techniques and multiple language solutions and expand our Forms applications significantly.

CHAPTER 5

Mixing and Mingling with the CLR

"Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."

— Donald Knuth

So far, we've skipped along the surface of the Common Language Runtime and the .NET framework, making use of features and functions without really examining the tools at hand to understand what's happening behind the scenes. This is the point where IronPython really begins to separate from the crowd and show off a bit, mixing the best of the Python language with the power and stability of the .NET framework.

"CLR-ance, Clarence."

What exactly is the Common Language Runtime? The CLR is a core component of the .NET framework, responsible for taking intermediate bytecode from the CIL, or Common Intermediate Language, and translating it to native code that can be run on the target platform. In plain English, you write your program source code in IronPython. Then this code is compiled to a standardized intermediate language (CIL). But this code isn't yet *native code,* that is, at a point where it can be executed by the operating system. The CLR takes care of that remaining step, converting the intermediate code to a final product that the operating system can interpret and execute.

In the previous chapter we saw how IronPython connects in code to the CLR, and we have already gotten some experience with standard .NET controls and events. What we need to do now is to take that knowledge a step further and build a completely functional, real-world application from the ground up. This lets us cover most of the phases of the **software development life cycle** and provide some concrete examples of how to work with the .NET framework.

■**Note** The *software development lifecycle*, or *SDLC*, is a series of steps that developers go through while working on software. There is no precise definition of these steps. Rather, you can consider five key areas involved in development: planning a project, gathering application requirements, designing the application, implementing that design, and, finally, maintaining that design. By definition this process tends to repeat, and some design methodologies break down individual steps into smaller, more discrete steps. But for the purpose of this discussion, these five are sufficient.

Cliché though it may be, I have a process for learning a new language that I'm going to share with you here, because it has served me well in the past. I find that it's easier to learn a language when I have something in mind to build; aimlessly toying with a method here or a pattern there doesn't really tie things together the way I'd like. Normally I build a small application from start to finish. It has to be an application that covers a variety of operations so that I can examine common (and occasionally not-so-common) aspects of the language. It has to be small enough that the project is manageable and not overly time-consuming. And finally, it has to have a small enough operational footprint that I don't have to implement a massive number of features or understand a complex problem domain to build it.

For our small application, we're going to build Notepad, and the CLR is going to help us do it.

■**Note** I hope the developer or developers at Microsoft who created Notepad don't get the impression that I'm marginalizing their work. If Notepad were poorly written, nobody would use it. You guys did great work, but I'd rather build Notepad from scratch in IronPython than build Windows Media Player, thank you very much! Oh, and if the *nix "vi" users could stop snickering we'd all appreciate it.

The Plan

Like any great bank heist (or piece of software, if crime isn't your thing), we have to formulate a plan. We already described some basic requirements for the application design itself, but we haven't addressed the functional requirements.

1. The application should consist of a single form.
2. The application should be capable of CRUD operations to the file system.
3. The application should operate on plain text documents.

4. The application should be written in pure IronPython; no other languages are allowed!
5. The application should be capable of printing documents.
6. The application should behave as much like Notepad as possible, wherever possible, although it need not implement every single aspect of the original program.

■ **Note** *CRUD,* an abbreviation stolen from the database world, stands for "create, read, update, and delete." Notepad does not support any functions that delete files from the file system, so we won't be implementing any here. Within reason we want our application to look and act as close to Notepad as possible.

The Design

If at all possible, it helps to pin down the design of an application early, with the understanding that virtually nothing is ever carved in stone. Indeed, many development teams find that a little design flexibility goes a long way, particularly when software maintenance comes up; an inflexible architecture is a difficult-to-maintain architecture. We'll see examples throughout the construction of our applications where changes are made significantly easier by anticipating, not the changes themselves, but the very fact that change is a constant in software engineering. That said, our text editing application should be fairly straightforward, but let's assume that it might evolve one day into a multimillion-dollar application and hedge our bets now.

In terms of the user interface, our application will be pretty straightforward. We'll have the primary form that houses all our controls, and we'll have a text box that essentially covers that entire form, except for the menu bar. The menu bar will be at the very top of the form to facilitate the file and print operations the user will need.

On the coding side, we'll have our main project that contains the form and necessary startup code. We will create a business logic folder and put all our code relating to program operation in there. As your applications grow, they can have as many layers as you consider appropriate, based on the needs of the application you're developing.

■ **Note** As you probably remember from Chapter 3, code that is tightly *coupled* is highly dependent on other code to do its job. We don't want that. Step 1 in the process of decoupling code (or, rather, ensuring it is never coupled in the first place) is making sure you don't throw everything into the code of the form itself. It makes maintenance a nightmare and induces a form of "code rigor-mortis."

The Implementation

The first thing we should do is set up a proper location for our development efforts. In Chapter 2 we set up C:\Python as the home for our IronPython code, so we'll stick with that. Open up IronPython Studio; we're going to create a new Forms application called *NotQuitePad* in the C:\Python\NotQuitePad directory, which we'll let IronPython Studio create for us (Figure 5-1). Make sure the "Create directory for solution" box is checked before you click the OK button; if it is not selected, then all the code will live at C:\Python instead of in a convenient subfolder.

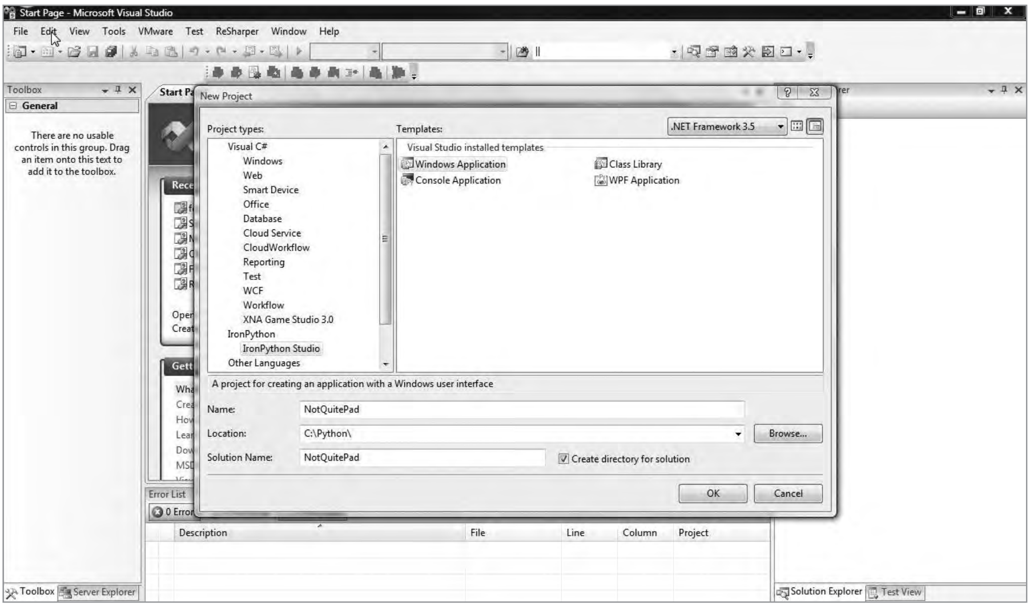


Figure 5-1. *Creating the NotQuitePad project*

When we created out project, IronPython Studio was gracious enough to generate automatically a form called *Form1*, which saves us from having to do it. Recall that earlier I mentioned that we should almost always change the names of controls we're going to be referencing in code because the default naming convention isn't the most meaningful scheme for a developer to interact with. Double-click on *Form1* in the Solution Explorer on the right; you should see the Properties window below it update with a variety of properties related to our form. Scroll to the top and change the property called (Name) to *NotQuitePad*. Do the same to the Text property.

In addition to the control names' not being particularly meaningful, Form1 isn't exactly a terrific description of our form either. Right-click on the *Form1.py* listing in the Solution Explorer and select Rename. Change the name to *NotQuitePad.py* and press Enter. Once you've done these steps, your screen should look similar to Figure 5-2.

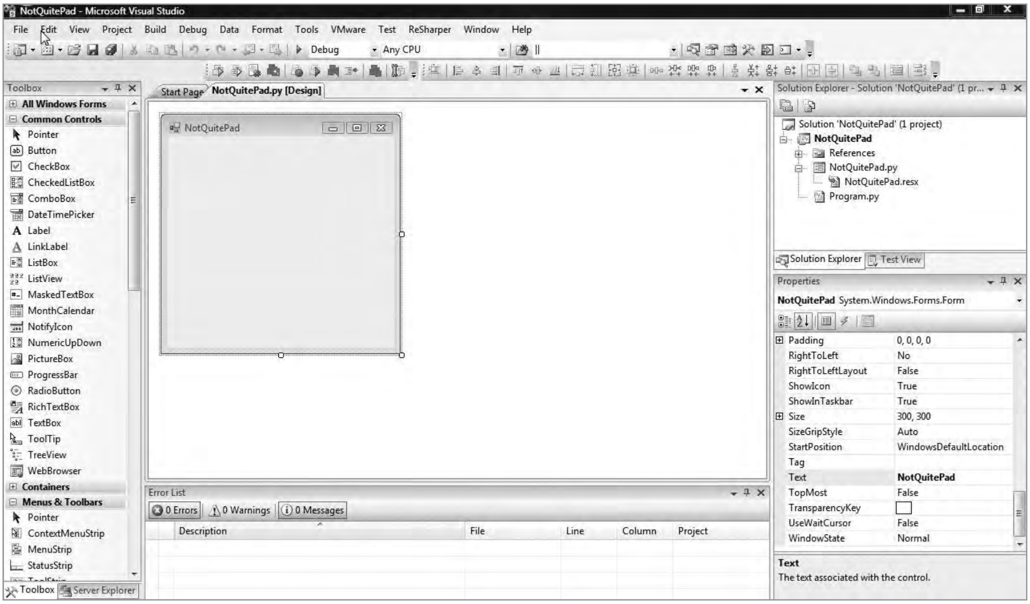


Figure 5-2. *NotQuitePad with a few settings tweaked*

Press F5 to build and debug the application. You should find that it doesn't execute the way you intended; the IDE should throw a fit about some code in *Program.py* (Figure 5-3).

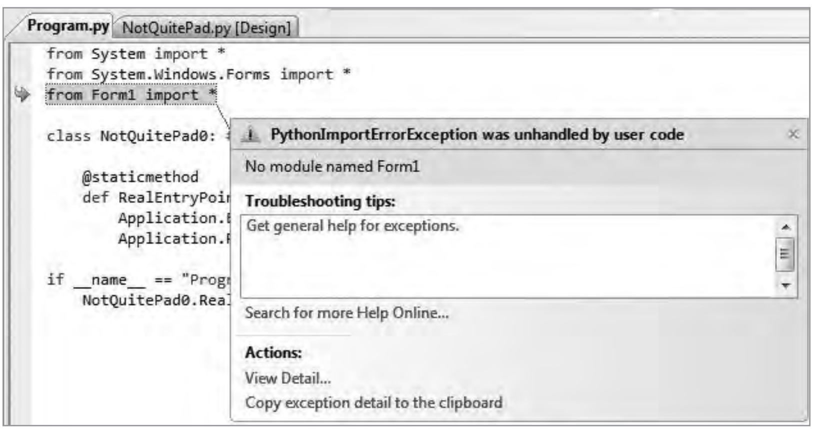


Figure 5-3. *Who were we calling again?*

I purposely didn't correct this error before having you run the application, to highlight a little "gotcha": the IDE won't update those code references automatically if you rename a form. Right now that's not a huge issue because we've only got one form in the

application and it's referenced in only one place. But you should definitely keep that in mind, because those types of errors can be insidious in large applications where certain forms are hit infrequently. Correct the code in *Program.py* so that it looks like Listing 5-1 (I have put the relevant changes in boldface, for convenience). Then try running it again (Figure 5-4).

Listing 5-1. *Fixing the Program.py Code*

```
from System import *
from System.Windows.Forms import *
from NotQuitePad import *

class NotQuitePad0: # namespace

    @staticmethod
    def RealEntryPoint():
        Application.EnableVisualStyles()
        Application.Run(NotQuitePad.NotQuitePad())

if __name__ == "Program":
    NotQuitePad0.RealEntryPoint();
```



Figure 5-4. *It's alive!*

Now that our application is starting to take shape, we have to add to the form the controls needed to operate the program. First up, we should drag to the form a text box control from the toolbox (Figure 5-5). Having done so, we rename the control to *txtUserText*. Scroll the Properties window until you see the property called Multiline; set

it to True. If you do not set the Multiline property to True, you will be able to resize only the control's width, and text cannot move down to a new line.



Figure 5-5. *Not quite Notepad size, but a start*

One of our application requirements is that the text box take up the entire form. We can do that through the properties, but let's do it through code instead. Double-click on the form at any location that is outside the boundaries of the text box. Immediately you should be taken to the source code for *NotQuitePad.py*. What you'll notice is that the IDE created a bit of code for you (Listing 5-2).

Listing 5-2. *The IDE Created a _NotQuitePad_Load Method for You Automatically*

```
@accepts(Self(), System.Object, System.EventArgs)
@returns(None)
def _NotQuitePad_Load(self, sender, e):
     pass
```

The IDE also wired an event for you that executes (or, in programmer lingo, *fires*) the *_NotQuitePad_Load* method when the form is loaded (Listing 5-3).

Listing 5-3. *The IDE Also Wired an Event to Fire When the Form Is Loaded*

```
…
self.Load += self._NotQuitePad_Load
```

■**Note** There's something really interesting at work here. Notice the usage of += to fire the *_NotQuitePad_Load* event when the form is loaded. If the IDE has just written an equals sign, the *only* thing to execute would be the method to the right of the equals sign. But since the IDE used a plus sign and an equals sign, our *_NotQuitePad_Load* method is part of a list of methods that can be run when the event is fired. This is incredibly powerful and will soon become very important to our application.

When the form loads, we can programmatically set the height and width of the *txtUserText* control to be equal to the height and width of the window. Remove the `pass` line of code and replace it with the code in Listing 5-4.

Listing 5-4. *Forcing the txtUserText Control to Be a Given Height and Width*

```
self._txtUserText.Size = self.Size
```

If you run the application again by pressing F5, you'll see that the *txtUserText* control is now the same size as the *NotQuitePad* form itself (Figure 5-6). However, things are not quite perfect yet.



Figure 5-6. *Well, it did exactly what we told it to.*

A good first attempt at fixing this problem might come in the form of modifying the method that fires when the application is loaded to, say, "Hey, put this TextBox as far to the top and left as you can get it!" This should put things in the right location. Let's do that right now. Modify your code to add the location definition shown in Listing 5-5 right after the size definition and rerun the application (Figure 5-7).

Listing 5-5. *Forcing the txtUserText Control to Move to the Top Left of the Form*

```
self._txtUserText.Location = Point(0, 0)
```

■**Note** A *Point* in .NET refers to an *X*-and-*Y* point in two-dimensional space, in that order, measured in pixels and starting from the top left of the containing object. As you probably recall from math classes, this is identical to the system used for plotting data, with one important distinction: in programming terms, the *Y*-value increases as it goes down the screen, not up as you might be used to. If we had set the location of the text box to be Point(10, 20), the text box would start 10 pixels to the right and 20 pixels *down* from the top left of the form. If that's a little confusing, feel free to play with the numbers and come back once you see how the coordinate space works; the bottom line is that *X* increases to the right on the screen and *Y* increases going down the screen.

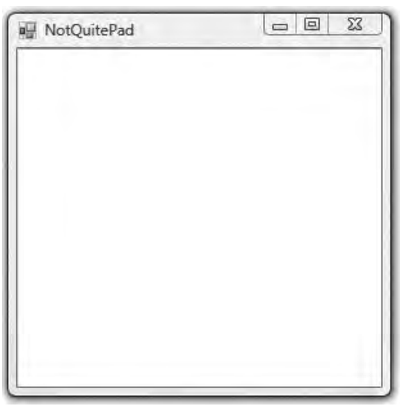


Figure 5-7. *Better! But looks can be deceiving.*

It looks like things are starting to take shape. But there's a problem. What happens when we resize the form? With the program running, drag the bottom right corner of the form down and to the right (Figure 5-8).

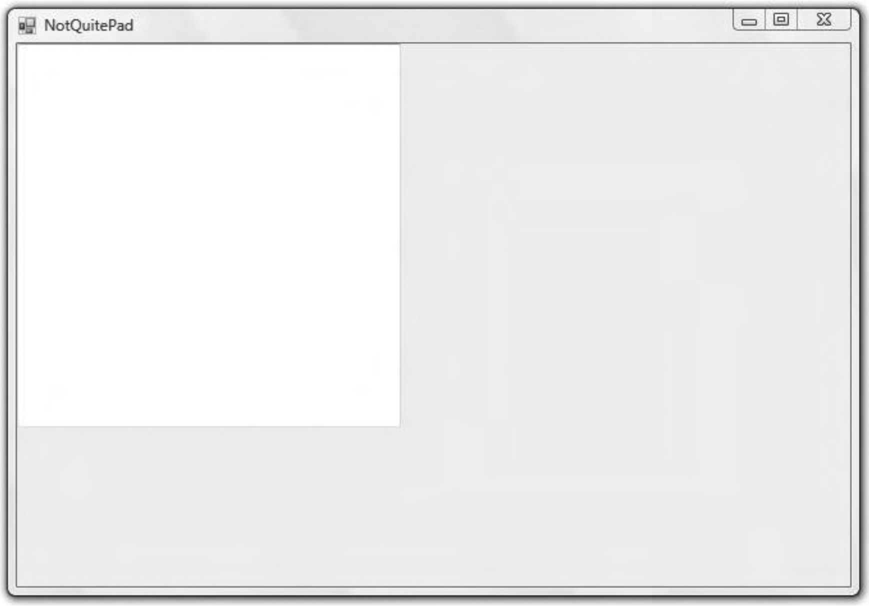


Figure 5-8. *Notepad would SO not do that.*

What exactly can we do about that? Clearly it's a deal-breaker to have the form be resized while the text box sits quietly at whatever size it was when the program was launched. Luckily the .NET programming model exposes a variety of events we can tap into. We've already seen this in action when the IDE added our *_NotQuitePad_Load* method to the list of events that fires when the form loads. We need to fire some code when the form is resized, so we will add an event to *self.Resize*.

We don't want to call the *_NotQuitePad_Load* method in our resize event; technically, it would work, but from a design and implementation standpoint it *absolutely stinks*. So instead, we'll make a method called *_ResizeInputBox* to handle the resizing and positioning of our text box when the user changes the size of the form. We should remove that resizing code from *_NotQuitePad_Load* and replace it with a call to our resizing method. It's a cardinal sin to duplicate code; it's always a best practice to implement a piece of code only once and to call it when needed.

■ **Tip** Here's one of these little "please don't shoot yourself in the foot" architecture moments where you can save yourself a terrible headache down the road. Consider this: Why is it better to write a single method that prints "Hello!" (let's refer to it as *SayHello*) and to call that method anywhere we need it, instead of adding a line of code that prints "Hello!" at each and every point we need it? If your boss comes to you six months from now and tells you, "Hey, we have to change the program so that it also prints the user's name in the greeting," you'll be kicking yourself for not making a simple method and using that instead. Hunting down code is no fun at all, and it is a particularly painful task when you're working under a deadline.

There are quite a few changes to the *NotQuitePad.py* file that can make this happen, so I'm going to provide the entire listing because it's easier when viewed in this fashion. Make sure your code matches what I've provided in Listing 5-6 before running the application again (Figure 5-9).

Listing 5-6. *The Complete Listing of NotQuitePad.py at This Time*

```
import System
from System.Windows.Forms import *
from System.ComponentModel import *
from System.Drawing import *
from clr import *
class NotQuitePad: # namespace

    class NotQuitePad(System.Windows.Forms.Form):
        """type(_txtUserText) == System.Windows.Forms.TextBox"""
        __slots__ = ['_txtUserText']
        def __init__(self):
            self.InitializeComponent()

        @accepts(Self(), bool)
        @returns(None)
        def Dispose(self, disposing):

          super(type(self), self).Dispose(disposing)
```

```
    @returns(None)
    def InitializeComponent(self):
        self._txtUserText = System.Windows.Forms.TextBox()
        self.SuspendLayout()
        #
        # txtUserText
        #
        self._txtUserText.Location = System.Drawing.Point(65, 115)
        self._txtUserText.Multiline = True
        self._txtUserText.Name = 'txtUserText'
        self._txtUserText.Size = System.Drawing.Size(100, 20)
        self._txtUserText.TabIndex = 0
        #
        # NotQuitePad
        #
        self.ClientSize = System.Drawing.Size(284, 264)
        self.Controls.Add(self._txtUserText)
        self.Name = 'NotQuitePad'
        self.Text = 'NotQuitePad'
        self.Load += self._NotQuitePad_Load
        self.Resize += self._ResizeInputBox
        self.ResumeLayout(False)
        self.PerformLayout()

    @accepts(Self(), System.Object, System.EventArgs)
    @returns(None)
    def _NotQuitePad_Load(self, sender, e):
        self._ResizeInputBox(sender, e)

    def _ResizeInputBox(self, sender, e):
        self._txtUserText.Size = self.Size
        self._txtUserText.Location = Point(0,0)
```

Caution If you've got eagle eyes, you may have noticed two things: (1) we haven't left room for the menu, and (2) when we call _ResizeInputBox in the *_NotQuitePad_Load* method, we aren't passing *self* like the signature seems to say we should be. On the first point, it's not important that we haven't left space for this element because we don't yet know how large it will be, and adjusting our starting Point values is a trivial matter. In terms of the method signature (point 2), remember that *self* is implied and added automatically. To prove this, add `self` before `sender` in the call to _ResizeInputBox and run the application. IronPython will tell you that you've provided one parameter too many.



Figure 5-9. *This is the behavior we specified for the application.*

Bad Medicine

Although the application is behaving properly, we've broken one of the rules we set for ourselves: we're dumping code into the form itself. If you hadn't noticed that we're doing that, then you'll have plenty of company; it's very easy for these bad habits to creep in when you're moving quickly or working on something unfamiliar and your focus is not on the overall organization of the code. If you did notice it, I hope you took the initiative to straighten things out! The rest of us will clean up the mess before moving on.

In the IDE, right-click on the *NotQuitePad* project, which appears as the first element in the *NotQuitePad* solution, and click Add and then New folder. Name this folder *business* and press Enter. Right-click on the *business* folder and click Add and then New Item. We want to add a class to our project that contains the code that resizes the form; we don't yet know what sort of future development the application will see, so we'll call this class *Interface.py* to indicate that code relating to the user interface is contained within. Press Enter, and IronPython will create the file containing the code in Listing 5-7.

Listing 5-7. *Our Interface.py Class, Brand-New to the World*

```
class interface:
    "Description of class"
```

Modify the *Interface.py* file so that your code looks like Listing 5-8. Feel free to rewrite the **docstring** if you choose; I would recommend doing so.

Listing 5-8. *Interface.py Updated*

```
from System import *
from System.Windows.Forms import *
from System.Drawing import *

class interface:
    "Code related to the user interface logic of NotQuitePad"

    def MoveInputBox(self, box):
        "Changes the location of an object to the origin (point 0, 0)"
        box.Location = Point(0, 0)

   def ResizeInputBox(self, box, windowSize):
        "Resizes an object to equal the size of another object"
        box.Size = windowSize
```

■ **Note** *Docstrings* are essentially a type of code documentation. Earlier in the book we covered the *help* function in IronPython, which you can employ to retrieve information about how to use a piece of code; any block of code that provides a docstring will have that string presented to the user when he or she pulls up the *help* information for it.

It's worth pointing out that, by definition, we've made some implicit design decisions about our code, and we need to walk through them. First off, because we'll be using some of the features provided by the .NET framework, we've begun by importing the relevant modules. How do we know which modules we'll need? The best source for information is the Microsoft Developer Network website at `http://msdn.microsoft.com/en-us/`, which provides in precise detail every method, namespace, and parameter in the framework. Bookmark it and consider it your lifeline to the deeper parts of the framework.

We have created two methods: *MoveInputBox* and *ResizeInputBox*. Note that these are good, descriptive names that make it pretty clear what function each serves. But by their naming convention, we're tying them to a specific purpose (that is, resizing a specific type of object.) Optionally, we could have named them something more general, such as *MoveBox* or *ResizeBox*. It's worth noting that decisions like this can be very subtle until you're used to building and working with application programming interfaces, or APIs. As you become more experienced you'll develop a sense of when to make methods specific and when to keep them more generalized.

The *MoveInputBox* method takes an object called *box* as a parameter. The only purpose of this method is to set the *Location* parameter of the object. In IronPython,

every object is **passed by reference**, not by value, so this change is reflected back in the calling method, and the text box is immediately placed at the origin, which is point 0,0 in the form.

■ **Note** In programming terms, objects can be passed by reference or by value. So if I pass an object by reference, when program execution returns to me, the initial object will potentially have been modified. If I pass an object by value, the original object is left completely unchanged.

Likewise the *ResizeInputBox* takes an object called *box* as a parameter, but additionally it takes a parameter called *windowSize*. This parameter will hold the value of the *Size* parameter of the form itself; the single purpose of this method is to set the size of the text box to that *Size* value. If you remember that IronPython passes objects by reference, it should be obvious that this change in size will immediately be reflected back on our form.

■ **Tip** Now is a good time to mention a very important concept in application development: *refactoring*. Refactoring is the practice of cleaning up the underlying structure of a program without changing the functionality or appearance of the program; if you're refactoring an application, you shouldn't be adding a whiz-bang new feature. It's a bit like cleaning up a room, with the intention of buying new furniture. The cleaner the room, the easier it is to add that swanky sofa you've been looking at for so long.

Technically, we could refactor our code and perform a little under-the-hood cleanup; the single responsibility principle says that a class should have one and only one reason to change. Our *interface* class technically has two reasons to change: (1) setting the location of the text box and (2) resizing it. We could refactor this one class into two classes, one with the responsibility for moving objects on a form and one responsible for resizing objects.

Now that we have designed the code, it's time to wire the interface so that our code can be called correctly during the program's event loop. The first thing we should do is delete any existing resizing code, restoring the application to the state of Listing 5-9. We want to eliminate the existing resizing code to reduce the potential for errors, plus in a moment it's going to be obsolete anyway because we've decided on a better method for achieving the same result.

■ **Note** It's not always necessary to "clean sweep" your code like this; in fact, many times it will be impractical. However, we haven't done much to this particular file, so it's a pretty easy revision to make.

Listing 5-9. *NotQuitePad.py Restored to the Initial State*

```
import System
from System.Windows.Forms import *
from System.ComponentModel import *
from System.Drawing import *
from clr import *
class NotQuitePad: # namespace

    class NotQuitePad(System.Windows.Forms.Form):
        """type(_txtUserText) == System.Windows.Forms.TextBox"""
        __slots__ = ['_txtUserText']
        def __init__(self):
            self.InitializeComponent()

        @accepts(Self(), bool)
        @returns(None)
        def Dispose(self, disposing):

            super(type(self), self).Dispose(disposing)

        @returns(None)
        def InitializeComponent(self):
            self._txtUserText = System.Windows.Forms.TextBox()
            self.SuspendLayout()
            #
            # txtUserText
            #
            self._txtUserText.Location = System.Drawing.Point(65, 115)
            self._txtUserText.Multiline = True
            self._txtUserText.Name = 'txtUserText'
            self._txtUserText.Size = System.Drawing.Size(100, 20)
            self._txtUserText.TabIndex = 0
            #
            # NotQuitePad
            #
            self.ClientSize = System.Drawing.Size(284, 264)
            self.Controls.Add(self._txtUserText)
            self.Name = 'NotQuitePad'
            self.Text = 'NotQuitePad'
            self.ResumeLayout(False)
            self.PerformLayout()
```

```
    @accepts(Self(), System.Object, System.EventArgs)
    @returns(None)
    def _NotQuitePad_Load(self, sender, e):
        pass
```

We know that duplicating code encourages code rot (and believe me, code *does* rot), and furthermore we know that the code in our *interface* class needs to execute *at least* once but likely *n* times: once when the application loads so that the text box is at the origin and resized to meet the window, and then an unknown number of times afterward as the form is maximized, minimized, and generally shuffled about the screen. The best way to handle a situation like this is *not* to copy and paste code all over the place, but to hand the responsibility for calling your *interface* class over to an intermediate method so that changes are limited to one area. That idea might sound a little confusing, but I think once you've seen it applied it will be clearer why we should always strive for this sort of design. We'll revise our application to Listing 5-10.

Listing 5-10. *NotQuitePad.py Revised to Minimize Future Maintenance*

```
import System
from System.Windows.Forms import *
from System.ComponentModel import *
from System.Drawing import *
from clr import *
from Interface import *
class NotQuitePad: # namespace

    class NotQuitePad(System.Windows.Forms.Form):
        """type(_txtUserText) == System.Windows.Forms.TextBox"""
        __slots__ = ['_txtUserText']
        def __init__(self):
            self.InitializeComponent()

        @accepts(Self(), bool)
        @returns(None)
        def Dispose(self, disposing):

            super(type(self), self).Dispose(disposing)
```

```
    @returns(None)
    def InitializeComponent(self):
        self._txtUserText = System.Windows.Forms.TextBox()
        self.SuspendLayout()
        #
        # txtUserText
        #
        self._txtUserText.Location = System.Drawing.Point(65, 115)
        self._txtUserText.Multiline = True
        self._txtUserText.Name = 'txtUserText'
        self._txtUserText.Size = System.Drawing.Size(100, 20)
        self._txtUserText.TabIndex = 0
        #
        # NotQuitePad
        #
        self.ClientSize = System.Drawing.Size(284, 264)
        self.Controls.Add(self._txtUserText)
        self.Name = 'NotQuitePad'
        self.Text = 'NotQuitePad'
        self.Load += self._NotQuitePad_Load
        self.Resize += self._ResizeFormEvent
        self.ResumeLayout(False)
        self.PerformLayout()

    @accepts(Self(), System.Object, System.EventArgs)
    @returns(None)
    def _NotQuitePad_Load(self, sender, e):
         self._HandleResizing()

    def _ResizeFormEvent(self, sender, e):
         self._HandleResizing()

    def _HandleResizing(self):
         newDimensions = interface()
         newDimensions.MoveInputBox(self._txtUserText)
         newDimensions.ResizeInputBox(self._txtUserText, self.Size)
```

Caution IronPython cares about the little space between parameters. If they're not separated by a space, it'll complain.

If you run the *NotQuitePad* application now, you should see something like Figure 5-10. If you resize the window, you should see that the text box not only remains anchored to the top left of the form, but also resizes to become the entire size of the form. If not, ensure your code matches what we've created so far and run the application again.



Figure 5-10. *Our refactoring effort was successful; we cleaned the code without changing behavior.*

I'd Like to See a Menu

The next step in our implementation is creating the menu bar. We don't need to have all the functionality at one time; we can start simply by getting the menu bar on the screen. With the main form open in Visual Studio, scroll down the toolbar options in the left-hand pane and expand Menus and Toolbars; then double-click MenuStrip. The menu will automatically appear on the top of the form in the typical location in which a menu normally appears in Windows. The menu will be blank except for a box that says, "Type here." Click that box and type **&File**. The ampersand (&) prefix allows the user to press Alt+F to open the File menu.

■ **Tip** It's important when designing menus to bear in mind which keys you have set as keyboard shortcuts so that you can avoid conflicts between hotkey settings. It's also a best practice to use shortcuts that are common and familiar to users of a given system. For instance, Windows users are typically quite accustomed to opening the File menu with Alt+F and to opening the Print dialog box with Ctrl+P. Don't fight the mental schema with which the average user is comfortable.

Below the File heading, go ahead and make a few more menu options; add "New", "Open…", "Save As…", and "Exit", which should leave you with a menu that looks like Figure 5-11.



Figure 5-11. *Placing some basic menu options*

Earlier I noted that when we wrote our code in the *interface* class to position the text box control at the origin (point 0,0 on the form), we didn't leave room for the menu because we didn't know how large it would be. If you run *NotQuitePad* now, you will see that although we've created a menu, it's not visible; the application still looks like it has just the one text box control on it. We need to find out how tall the menu is and then adjust our *interface* method.

Click the menu in the Visual Studio design window; then scroll through the Properties window in the bottom right until you see the Size property. The size of the menu is 284, 24 on my screen, which means we need to adjust our positioning code to start the text box *25 pixels down* from the origin. Before you do so, let's change the Name property of the menu from *menuStrip1* to *mainMenu* so that we can identify it easily.

■ **Note** Why 25 pixels instead of 24? The reason is that the menu strip itself is 24 pixels tall; if we move 24 pixels down the form, we'll actually cut off one horizontal row of pixels with our text box. It's a tiny issue that a lot of people actually miss.

Open *Interface.py* and change the *MoveInputBox* method to set the initial location to be Point(0, 25). If you've been including docstrings, make sure to update the content of it so that the description matches the implementation.

Listing 5-11. *Interface.py with the MoveInputBox Method Updated*

```
from System import *
from System.Windows.Forms import *
from System.Drawing import *

class interface:
    "Code related to the user interface logic of NotQuitePad"

    def MoveInputBox(self, box):
        "Changes the location of an object to the origin, leaving room for a ➥
menu(point 0, 25)"
        box.Location = Point(0,25)

    def ResizeInputBox(self, box, windowSize):
        "Resizes an object to equal the size of another object"
        box.Size = windowSize
```

Run the program again and check out your snazzy new menu (Figure 5-12). Remember to resize it a few times, and do a quick quality assurance test that everything is working as intended.



Figure 5-12. *The menu is in place and the text box is leaving room for it.*

Reading, Writing, Arithmetic

We've got a pretty good interface going. We've succeeded in building something that meets our requirements and bears more than a passing resemblance to its built-in Windows cousin. While there are more interface bells and whistles we could be adding right

now, we should take some time to wire back-end functionality together with the front end; as with any structure, the foundation needs to be strong before we concern ourselves with the aesthetics. We'll begin this task by creating a new *class* file in our *business* folder called *fileOperations.py*.

At the beginning of this chapter, we laid out our requirements for the application. The most critical one clearly is the Create, Read, and Update operations in the file system. Before we lay out the methods our class will need, it would be prudent to decide what our naming convention will be. We need to decide whether to stick with the names of this triad of methods or whether we should opt for method names that relate more closely to file operations. For right now, I'm going to opt for the standard file operation naming convention: New, Open, Save. Take a look at Listing 5-12 for an example of what I mean.

Listing 5-12. *The Initial Design of fileOperations.py*

```
from System import *
from System.IO import *

class fileOperations:
    "Contains file system operations for NotQuitePad."

    def New(self):
        "Creates a new file within NotQuitePad."
        pass

    def Open(self):
        "Handles the Open dialog window."
        pass

   def _OpenFileFromDisk(self, fileName):
        "Opens a connection to the file system for opening a file."
        Pass

    def Save(self):
        "Handles the Save dialog window."
        pass

    def _WriteFileToDisk(self, fileName, fileContents):
        "After executing the Save method, write the file and contents to the➥
desired location."
        pass
```

There's an interesting point to make here: we have method signatures that start with an underscore character. This is to indicate that we want this method to be "private." What we want to have happen is that when the user clicks "Save As" in the menu, it will execute the Save method, which should open that familiar Windows dialog box for saving a file. Once the file name and location have been chosen, we want that *Save* method to call an internal method called *_WriteFileToDisk*; we don't want code accessing this method in an incorrect order. Furthermore we may want additional steps in this process at a later date. If we allow *Save* to call a deeper, private method, then we add a layer of abstraction to the entire process and provide flexibility for unforeseen design changes. We're instructing ourselves and others not to call *WriteFileToDisk* directly. The same scenario applies to the *Open* and *_OpenFileFromDisk* methods as well.

Open Sesame

The task of displaying a functional dialog box that is suitable for opening or saving files is essentially boilerplate code. One of the great things about .NET is that it provides a huge library of boilerplate code snippets to do all of the basic Windows tasks. So not only is it simple to display Open and Close dialog windows, but they're also standardized in appearance so that users have little if any learning curve with our application's basic operations.

Let's start to use these code snippets, by learning how to open a file from disk. The dialog windows facilitate a common user experience and allow us easy access to a particular file the user has selected. This is best demonstrated with an example. Open *fileOperations.py* and modify it to look like Listing 5-13.

Listing 5-13. *FileOperations.py with a Basic Open File Dialog Window*

```
from System import *
from System.Windows.Forms import *
from System.Windows import *
from System.IO import *

class fileOperations:
    "Contains file system operations for NotQuitePad."

    def New(self):
        "Creates a new file within NotQuitePad."
        pass
```

```
    def Open(self):
        "Handles the Open dialog window."
        dialog = OpenFileDialog()
        dialog.Title = "Load File"
        if dialog.ShowDialog() == DialogResult.OK:
                pass

    def _OpenFileFromDisk(self, fileName):
        "Opens a connection to the file system."
        pass

    def Save(self):
        "Handles the Save dialog window."
        pass

    def _WriteFileToDisk(self, fileName, fileContents):
        "After executing the Save method, write the file and contents to the➥
desired location."
        pass
```

What's happening here is pretty straightforward. We are creating an instance of an *OpenFileDialog* object, setting the title to be *Load File*, and checking what button the user pressed. If the user pressed "OK," we'll eventually have code here to actually open the file and read its contents from disk. It may seem counterintuitive, but in .NET the *file open* and *file save* dialogs are a bit separated from the physical files themselves. They provide a method of interacting with the disk; developers need to make use of streams to actually write to or read from the disk. The word *stream* means there is one-way communication. If we are reading a stream of data from the disk, we start at the beginning and read an arbitrary number of bytes, in order.

Before we go mucking about in the stream, let's try out this dialog window. Go back to *NotQuitePad.py,* click File in your menu, and then double-click the Open menu option. IronPython Studio will automatically create a method called *_openToolStripMenuItem_Click* that fires when the Open menu option is clicked. Add to that method the code in Listing 5-14. Next, import the *fileOperations* class at the top of the file so that IronPython has a reference to the code you created. Now run the application (Figure 5-13).

Listing 5-14. *Calling Our Dialog Method from the Interface*

```
def _openToolStripMenuItem_Click(self, sender, e):
     openDialog = fileOperations()
     openDialog.Open()
```

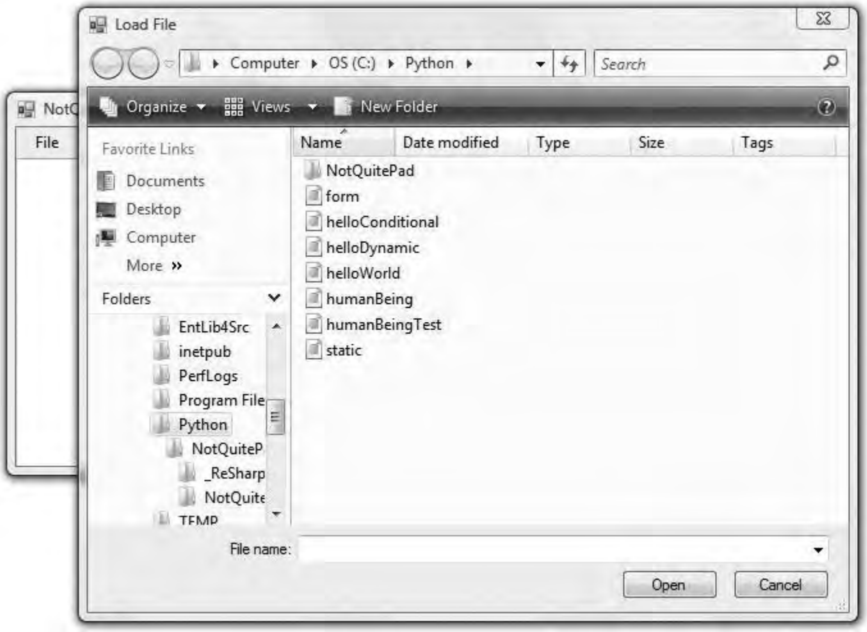


Figure 5-13. *Our dialog menu displays, but it does not automatically do anything with a selected file.*

Although the dialog window does display and we can select a file, when we open it nothing happens. The reason for this is twofold: (1) we have not wired our file-opening code to the interface code so that the file contents are displayed, and (2) we have not yet actually read the file contents from the disk. The first modification to make is in the *_openToolStripMenuItem_Click* method. Change the method to look like Listing 5-15.

Listing 5-15. *Populating the Text Box with Data, When Available*

```
def _openToolStripMenuItem_Click(self, sender, e):
     openDialog = fileOperations()
     self._txtUserText.Text = openDialog.Open()
```

Now we have provided a way to update the user interface with the contents of the file read from disk; the next step is to open a stream and read the contents of a given file that the user selects from the dialog box. Open *fileOperations.py* and modify it to look like Listing 5-16.

Listing 5-16. *Handling the Physical Read from the Disk*

```
from System import *
from System.Windows.Forms import *
from System.Windows import *
from System.IO import *

class fileOperations:
    "Contains file system operations for NotQuitePad."

    def New(self):
        "Creates a new file within NotQuitePad."
        pass

    def Open(self):
        "Handles the Open dialog window."
        dialog = OpenFileDialog()
        dialog.Title = "Open"
        if dialog.ShowDialog() == DialogResult.OK:
                contents = self._OpenFileFromDisk(dialog.FileName)
                return contents

    def _OpenFileFromDisk(self, fileName):
        "Opens a connection to the file system."
        file = File.OpenText(fileName)
        data = file.ReadToEnd().ToString()
        file.Close()
        return data

    def Save(self):
        "Handles the Save dialog window."
        pass

    def _WriteFileToDisk(self, fileName, fileContents):
        "After executing the Save method, write the file and contents to the➥
desired location."
        pass
```

Tip If you have any prior Python programming experience, you have probably noticed how strikingly similar the .NET method of opening a file is to Python's. You could substitute the following Python code in the *_OpenFileFromDisk* method and get the exact same result. Technically .NET will apply some formatting for you based on any nonprintable characters in the contents (such as line breaks), but the net result of accessing a file's contents is the same for a comparable amount of code.

```
file = open(fileName)
data = file.read()
file.close()
return data
```

If desired you can employ traditional Python methods anywhere you like in IronPython to do just about any task. But you may find that in doing so you have to write more code to perform basic boilerplate operations (think of creating a form on the screen, making a functional menu bar, and so on).

This is the proof in the pudding from earlier about separating the file-opening code from the dialog code. Now the workload is divided properly and cleanly; the *Open* method handles the dialog window itself and makes a call to *_OpenFileFromDisk* to do the heavy lifting and send the contents of the desired file back for return to the user interface code. If everything had been smashed together in the interface form code, you can imagine how quickly things would get bloated, even with the limited functionality we've introduced to a program designed at the Notepad level. Hopefully you're already seeing things in terms of ease of maintenance.

This section of code is our first low-level exposure to the *System.IO* namespace in .NET. This namespace exposes a lot of functionality for dealing with the file system in particular. The *File.OpenText()* method in this namespace is responsible for opening a connection to the file on the disk, and *.ReadToEnd()* reads the entire contents of that file until the EOF (end of file) marker, in a one-way fashion. The *File* class also exposes other methods that are used for reading individual lines or bytes from a file. We are calling the *.ToString()* method to convert the Stream contents to a data type that the text box can properly display.

Note Connections to the file system are expensive resources. If you open them, make sure you close them! The same applies to resources such as database connections. Nothing kills database performance (and application performance as a result) faster than not closing connections when you're through with them. We'll cover proper close and dispose patterns throughout the book, but the best advice is what my parents used to tell me: "Close the door! We're not heating the whole neighborhood here."

If you run the application now, click File, then Open, and then select a text file from your drive (an IronPython source code file works great in this case). I opened the source code to this particular file, and my screen looks like Figure 5-14.

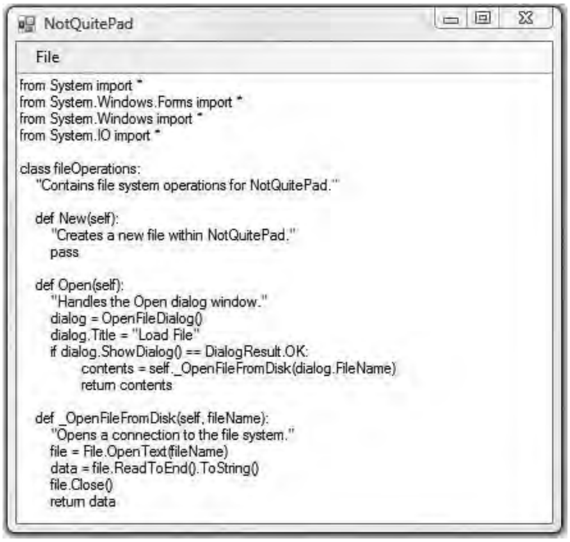


Figure 5-14. *With the file stream code in place, the Open function works.*

I Can't Even Save Myself

Having written IronPython code to open a text file from disk and to do a little work with streams, we now know enough to write code to save text back to the disk. But we have some additional considerations to take into account; a good example would be answering the question "If the destination file already exists, do we cancel or overwrite the destination file?" What we will do is ask the user if she or he wants to overwrite the existing file (if one does in fact exist); if so, we'll wipe out what's there and replace it with new content. If the user chooses not to overwrite, we will not append any content; we'll simply remain at the dialog window.

■ **Note** The only two choices we really have based on user input here are (1) to do nothing if the user cancels and (2) to overwrite the entire file with the new contents if the user chooses to save. You can always append to the file, meaning you could add additional text to the end of the existing file. But this behavior wouldn't gel with the way Notepad operates and would be contrary to what the typical end user expects of this type of operation, so we'll move forward with our two primary choices only.

Thankfully the .NET framework developers were nice enough to stick with a common naming convention; where we used *OpenFileDialog* to browse the file system and select a file, now we will use a *SaveFileDialog* to do the opposite and save a file back to the disk. On our end, we will stick to our own convention and separate the physical task of writing to disk from the interaction task of finding that location.

■ **Note** I think you'll love how terribly simple this next bit is. This is one of those moments when a decent design at the beginning pays off big dividends down the line.

First, let's open *fileOperations.py* and add the code in Listing 5-17 to the *Save* method. It should look pretty familiar; it's essentially identical to what we did in the *Open* method, just modified a bit to save a file instead of open one.

Listing 5-17. *Setting Up a Save File Dialog in fileOperations.py*

```
def Save(self, fileContents):
    "Handles the Save dialog window."
    dialog = SaveFileDialog()
    dialog.Title = "Save As"
    if dialog.ShowDialog() == DialogResult.OK:
            pass
```

Now return to *NotQuitePad.py*'s Design View, click on the File menu option, and then double-click Save As. As with the Open option, IronPython Studio creates a method signature to handle the option being clicked. We need to make an instance of our *fileOperations* class and call the *Save* method, passing the contents of the text box as a parameter (Listing 5-18).

Listing 5-18. *Calling the Save File Dialog from the User Interface*

```
def _saveAsToolStripMenuItem_Click(self, sender, e):
    saveDialog = fileOperations()
    saveDialog.Save(self._txtUserText.Text)
```

Run the application and give the Save As menu option a spin. As with the Open menu option, at this point in the process we haven't wired everything together, so, although the dialog menu will display, it won't yet save the file (Figure 5-15). Feel free to type some text and to try to save it to disk; you'll find that there are no errors but that the file doesn't get saved. We'll tackle that now.

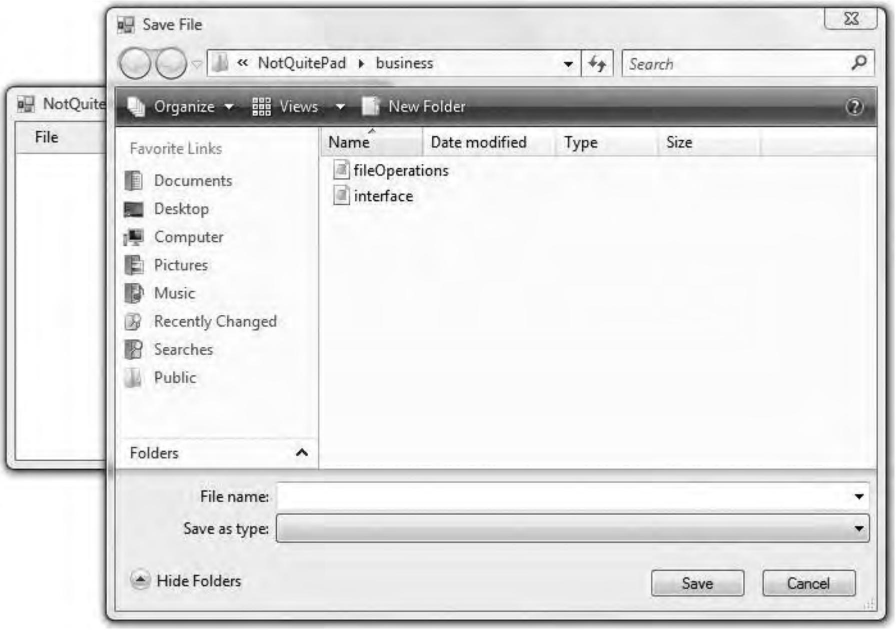


Figure 5-15. *History repeats; until we wire the UI to the business layer, this is for show only.*

Open *fileOperations.py* and modify it to look like Listing 5-19. The code for saving is again almost identical to what we did for opening the file.

Listing 5-19. *fileOperations.py with Save As Implemented*

```
from System import *
from System.Windows.Forms import *
from System.Windows import *
from System.IO import *

class fileOperations:
    "Contains file system operations for NotQuitePad."

    def New(self):
        "Creates a new file within NotQuitePad."
        pass
```

```
    def Open(self):
        "Handles the Open dialog window."
        dialog = OpenFileDialog()
        dialog.Title = "Open"
        if dialog.ShowDialog() == DialogResult.OK:
                contents = self._OpenFileFromDisk(dialog.FileName)
                return contents

    def _OpenFileFromDisk(self, fileName):
        "Opens a connection to the file system."
        file = File.OpenText(fileName)
        data = file.ReadToEnd().ToString()
        file.Close()
        return data

    def Save(self, fileContents):
        "Handles the Save dialog window."
        dialog = SaveFileDialog()
        dialog.Title = "Save As"
        if dialog.ShowDialog() == DialogResult.OK:
                     self._WriteFileToDisk(dialog.FileName, fileContents)

    def _WriteFileToDisk(self, fileName, fileContents):
        "After executing the Save method, write the file and contents to the➥
desired location."
        file = File.CreateText(fileName)
        file.Write(fileContents)
        file.Close()
```

So what's happening is that we're using a *StreamWriter* object named *file* to write the contents of our text box back to the file system, making sure to call the .*Close()* method at the end to clean up our resources. Press F5 to run the application again, create a document, and then verify that the Save As and Open features work as expected (Figure 5-16).

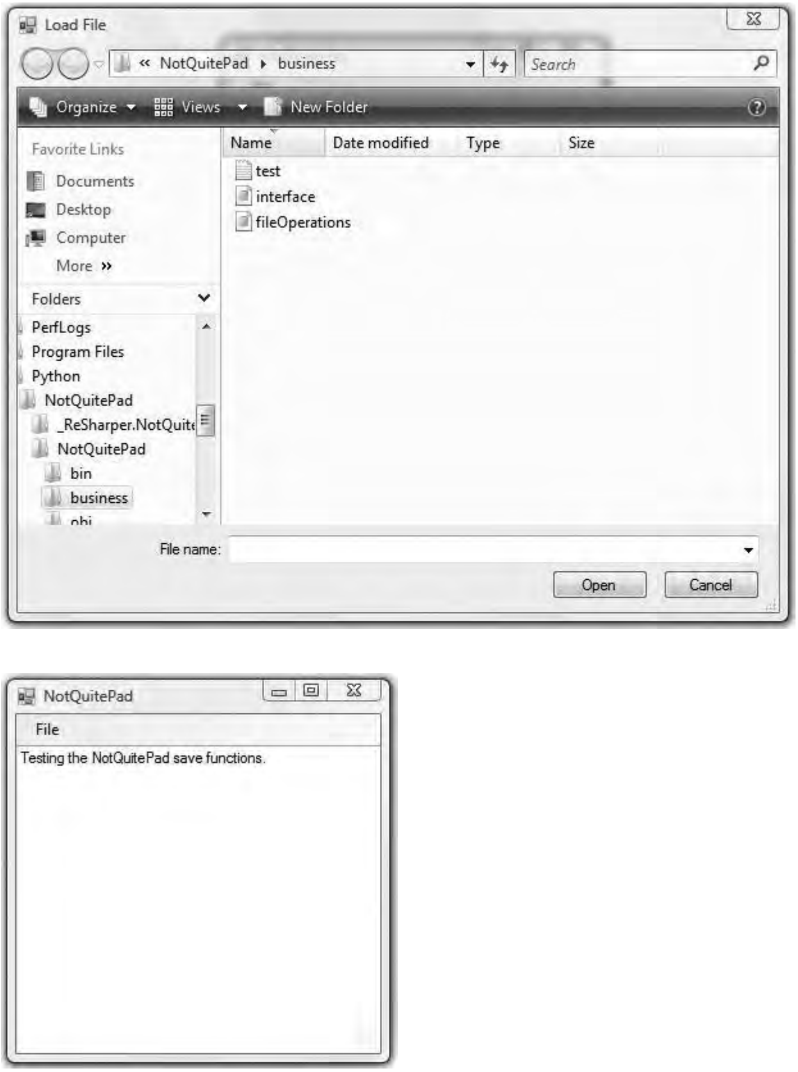


Figure 5-16. *The Save As functionality works; I have opened a file I created with NotQuitePad.*

Print, Please

Our users are spoiled; they actually like to *print* the documents they create. I suppose we can indulge them this one time. First things first. Let's head to the Design View for the main *NotQuitePad* form and add a Print option to the drop-down menu.

Click on the File option to open the menu, and you will see a blank box with the text "Type Here" below the Exit option. In that box, type **Print**, then press Enter. The Print option will be added to the menu, but it's at the bottom, which is not the location a user expects to find such. So left-click and drag it between Save As and Exit. Next, double-click that Print option so that we can wire some functionality to it.

The IDE will dutifully create a snippet of code for you to start with, which we have modified in Listing 5-20 to call our *fileOperations* class and to use a *.Print()* method we will create shortly that accepts our text box text as a parameter.

Listing 5-20. *The Code for the Print Option Menu Click*

```
@accepts(Self(), System.Object, System.EventArgs)
 @returns(None)
 def _printToolStripMenuItem_Click(self, sender, e):
        printDialog = fileOperations()
        printDialog.Print(self._txtUserText.Text)
```

Open up the *fileOperations* class. For the purposes of printing documents, we're going to use another type of dialog that .NET provides called the *PrintDialog*. It works in a very similar fashion to the dialogs we've used so far. Let's look at a complete implementation and break the code down (Listing 5-21).

Listing 5-21. *Calling the Print Dialog*

```
def Print(self, fileContents):
        "Handles the Print dialog window."
        dialog = PrintDialog()
        if dialog.ShowDialog() == DialogResult.OK:
            doc = PrintDocument()
            dialog.Document = doc
            self._PrintDocument(doc, fileContents)

 def _PrintDocument(self, printDocument, fileContents):
        "Sends a document to the selected printing device."
        printDocument.Print()
```

The *PrintDialog* expects a *PrintDocument* object to be passed to it, which we define as *doc*. The document is passed to our private *_PrintDocument* handler, which is responsible for any tasks related to the actual printing of the document (Figure 5-17).



Figure 5-17. *The Print Dialog in Action*

A Touch of OOP

There's one menu option we haven't touched yet, the New option, which creates a new document. We can do this in variety of ways, including just blindly deleting any text in the main text box, which is really not the most elegant way to do things. A better, more flexible approach would be to create a class that represents our document, with methods exposed, that lets the calling code know the status of the document's workflow.

In the *business* folder of your project, create a new class called *document.py*. To track whether or not a document has been modified since saving (if it has been saved at all), we're going to create two methods: *IsDirty()* and *SetDirty()*. These methods will allow us quickly to check the state of a document when needed to make decisions about what to display to the user (Listing 5-22).

Listing 5-22. *The Document Class Implementation*

```
from System import *

class document:
    "Represents a document in NotQuitePad."

    # properties of the document
    _isDirty = False
    _contents = String.Empty
```

```
    def IsDirty(self):
        "Gets whether or not a document has been saved since modification."
        return self._isDirty

    def SetDirty(self, value):
        "Sets whether or not a document has been saved since modification."
        self._isDirty = value
```

■ **Note** For right now, we're going to wire the document state-checking code to the New menu option only.

Open the main *NotQuitePad* form again in Design View, and double-click on the New menu option to create the click stub as we have done before. In the interest of application design, we should keep to the design patterns we've followed so far and leave as much code out of the interface as possible so that the user interface (UI) is not burdened with excessive decisions. Modify the click method as shown in Listing 5-23.

Listing 5-23. *Handling the New Menu Option*

```
@accepts(Self(), System.Object, System.EventArgs)
@returns(None)
def _newToolStripMenuItem_Click(self, sender, e):
     newDocument = fileOperations()
     if newDocument.New() == True:
          self._txtUserText.Text = String.Empty
     else:
          newDocument.Save(self._txtUserText.Text)
          self._txtUserText.Text = String.Empty
```

By calling down into the *fileOperations* class, the UI doesn't ever need to know (nor will it) whether a document's *_isDirty* flag is set to True or not. It only knows how to handle content based on abstract decision trees further in. What we've done here is say that if the *.New()* method returns True, we should clear the text box completely. This indicates that the file was not marked dirty and did not need to be saved. Otherwise, we need to display the *.Save()* dialog and give the user the option to save her work.

We need to establish a *document* object in the *fileOperations* class so that we can maintain the document workflow state throughout the application. At the top of the *fileOperations* class, add the line shown in Listing 5-24 immediately below the *.New()* method.

Listing 5-24. *Creating a Document Object for the Class to Use*

```
doc = document()
```

The next step is to fill out the *.New()* method we created so long ago in *fileOperations.py*. The implementation of it is shown in Listing 5-25.

Listing 5-25. *Fleshing Out the New Command in fileOperations.py*

```
 def New(self):
        "Creates a new file within NotQuitePad."
        if self._CheckIfFileIsDirty() == True:
                        msg = MessageBox.Show("Do you want to save the changes ➥
to your file?", "NotQuitePad", MessageBoxButtons.YesNo)
        if msg == DialogResult.Yes:
                         return False
                    else:
                         return True
         else:
                    return True

 def _CheckIfFileIsDirty(self):
        "Call the document class to find out if a document has been marked dirty➥
and needs to be saved."
        return self.doc.IsDirty()
```

The first thing worth noticing is the private method *_CheckIfFileIsDirty()*, which calls down into the *document* class to find out if a document is marked dirty. We specifically want to keep that sort of functionality *out* of the *.New()* method; it's best that any functionality related to the document workflow checking be kept separate, in case it changes in the future. The *.New()* method is called when the New menu option is clicked. It immediately checks whether a document has been marked dirty, and if so it creates a *MessageBox* object that asks the user if he wants to save changes to the file. If so, the code returns False back to the UI to indicate that the method determined the file was not ready to be saved and the user wishes to save it. If the file wasn't dirty in the first place, the method returns True to indicate that the text box contents can be cleared.

■ **Note** The *MessageBox.Show()* method has several different overloads that expose different options. In this case, we're using the overload with parameters (String, String, MessageBoxButtons). For more information, see `http://msdn.microsoft.com/en-us/library/system.windows.forms.messagebox.show.aspx`.

Next, we need a way to tell the document class that our document should be marked dirty. To do so, open the main form in Design View and double-click on the text box. This should create a stub called *_txtUserText_TextChanged()*. This event is fired every time the contents of the text box change; we will make a call to our *fileOperations.py* class so that the UI remains ignorant of document code (Listing 5-26).

Listing 5-26. *The TextChanged Event Makes a Call to the fileOperations Class*

```
@accepts(Self(), System.Object, System.EventArgs)
@returns(None)
def _txtUserText_TextChanged(self, sender, e):
     file.SetDirty(True)
```

Next, we need to implement the *.SetDirty()* method in *fileOperations.py* (Listing 5-27).

Listing 5-27. *SetDirty Implementation in fileOperations.py*

```
 def SetDirty(self, value):
        "Call the document class to set the dirty property of a document."
        self.doc.SetDirty(value)
```

The last step in this process is modifying the UI code so that the *fileOperations* class knows to set the Dirty flag to False after a successful creation of a new document (Listing 5-28).

Listing 5-28. *Letting the UI Know to Pass Along That a Document Is No Longer Dirty*

```
@accepts(Self(), System.Object, System.EventArgs)
@returns(None)
def _newToolStripMenuItem_Click(self, sender, e):
     newDocument = fileOperations()
     if newDocument.New() == True:
          self._txtUserText.Text = String.Empty
     else:
          newDocument.Save(self._txtUserText.Text)
          self._txtUserText.Text = String.Empty
newDocument.SetDirty(false)
```

Exit Strategy

You've done all the hard work of getting a basic version of Notepad running; luckily, getting it to shut down is much simpler. Although it's tempting to drop the code for exiting the application into the UI because it is terribly short, we should stick to our design patterns and move it to an abstracted location. The best home for this code is in the *Interface.py* file with the other UI code we created earlier in the chapter. Add the method shown in Listing 5-29 to that file.

Listing 5-29. *Exiting an Application*

```
def AppExit(self):
     "Exits the current application"
     Application.Exit()
```

Now we can call this method from the Exit command in the drop-down menu in the main form (Listing 5-30).

Listing 5-30. *Calling the Custom Exit Code in the interface Class*

```
def AppExit(self):
     ui = interface()
     ui.AppExit()
```

■ **Note** This application exit code is a prime candidate for adding an *.IsDirty()* check as a user convenience.

Beautification

One of the last steps in our NotQuitePad wrap-up is to provide a prettier typeface. Open the main form in Design View and click on the text box. Set the Font to be Lucinda Console size 10, if you have that font on your machine; if not, pick one of your preferences. Next, we need to set the vertical scrollbars to appear when the text gets too large for the display window, so scroll to the Scrollbars property to Vertical.

Recall that in our *interface* code, we had set the text box width and height to be the same as those of the form. The problem with that now is that we can't see the scrollbar (feel free to run the application yourself real quick). Let's open the *Interface.py* file and make some adjustments to the resizing code to accommodate the scrollbar (Listing 5-31, Figure 5-18).

Listing 5-31. *Leaving Room for the Scrollbars*

```
def ResizeInputBox(self, box, windowSize):
     "Resizes an object to equal the size of another object"
     windowSize.Height = windowSize.Height - 60
     windowSize.Width = windowSize.Width - 15
     box.Size = windowSize
```

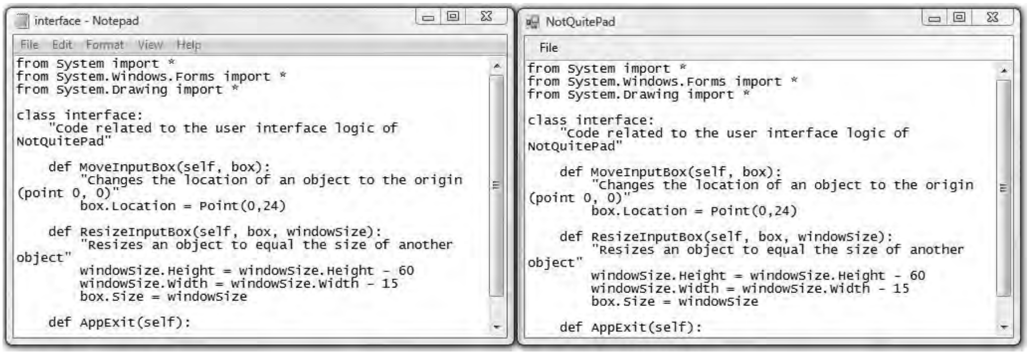


Figure 5-18. *Side-by-side comparison*

Project Postmortem

Well, hopefully that wasn't too painful! Developing IronPython applications is actually straightforward and logical. Now that we've got a functional application, we should review the plan requirements from earlier in the chapter and see how we did.

1. The application should consist of a single form. **DONE**.

2. The application should be capable of CRUD operations to the file system. **DONE**.

3. The application should operate on plain text documents. **DONE**.

4. The application should be written in pure IronPython; no other languages allowed! **DONE**.

5. The application should be capable of printing documents. **DONE**.

6. The application should behave as much like Notepad as possible, wherever possible, although it need not implement every single aspect of the original program. **DONE**.

Although we did not implement every single feature of Notepad, we did succeed in fulfilling the requirements we set for ourselves at the beginning of the chapter, and the application is stable enough for some initial testing.

Summary

From design patterns to concrete implementations, in one chapter we went from empty project to basic text editor. We looked at object-oriented programming concepts, wrote code that we could easily extend in the future, and saw how IronPython hooks into the .NET framework to extend the functionality of both Python and .NET itself.

CHAPTER 6

Advanced Development

"The cost of adding a feature isn't just the time it takes to code it. The cost also includes the addition of an obstacle to future expansion. The trick is to pick the features that don't fight each other."

— John Carmack

Up to this point, we've built fairly linear applications that are not truly component-based. Although we designed with future maintenance in mind, we haven't built with the idea that we may want to reuse parts of our application later in related (or totally different) applications. We would be doing ourselves a disservice if we didn't take a look at how to do this sort of development in IronPython.

Note The primary focus of this chapter is going to be interacting with C#, another language in the .NET family. We'll be looking at how to extend an application using IronPython as a plug-in architecture, and some of the C# syntax is fairly advanced. Don't get too hung up about the C#; follow the directions and enter the code exactly as it appears. The important thing here is to examine how a statically typed language like C# can interact with a dynamically typed one like IronPython. On the other hand, if you're familiar with C# (or you want to learn about it), then hopefully this will introduce you to some concepts that (I think) are pretty darn cool.

Base Classes for Fun and Profit (aka "The LEGOs on the Bottom Don't Really Exist")

One of the conveniences of object-oriented software development is polymorphism, which we briefly covered much earlier in the book. *Polymorphism*, in simplest terms, is the ability of an object of one type to be substituted seamlessly for an object of another type. In IronPython this sort of functionality is easily achieved; indeed, most modern languages actually support this. However, the difference between a dynamic language and

a static one is in how this is actually implemented. A dynamic language performs polymorphic operations implicitly based on which methods are present in a class, whereas a static language requires explicit typing information and either an abstract-base-class implementation or interface implementation.

Whew! That hits you like reading stereo instructions, doesn't it? A little code might help clarify matters a bit. Let's start with C# so that we can see how a statically typed language handles polymorphism (I promise the detour to C# is temporary but useful; we'll get back to pure IronPython in the next chapter). Assume for a moment that for our current application we're going to need to implement some basic object that represents a human being (we could go deeper and start with "living thing," but this is sufficient for our purposes). In C#, we have a few choices on how to do this. One way is via an **abstract base class**. Any subclasses, to compile properly, have to inherit and implement fully the members of the base class.

■**Note** An *abstract base class* is a class that cannot be instantiated but is only used for other classes to inherit from, hence the term *abstract*. It generally serves as a starting point for a variety of subclasses that need to implement some common methods.

Let's open up Visual Studio or the free Microsoft Visual C# 2008 Express Edition. We're going to create a console application, but using C# (Figure 6-1). Under File ➤ New Project, select the console application. Call this application. I'm placing this in my *C:\Python* directory for the sake of keeping our projects together. You could put it in another location, if you prefer; it won't hurt anything to do so. Save the project before continuing.

Once you've created the project, you'll be presented with a screen similar to the one in Figure 6-2. Although I don't expect you to know C#, what should stand out is that both the application file structure and the code itself bear a close similarity to the structure of programs we've written in IronPython. The initial C# file that's created adds a few references to .NET namespaces and defines a class and a method with some parameters. Syntactically there are differences, but the underlying tasks are the same.

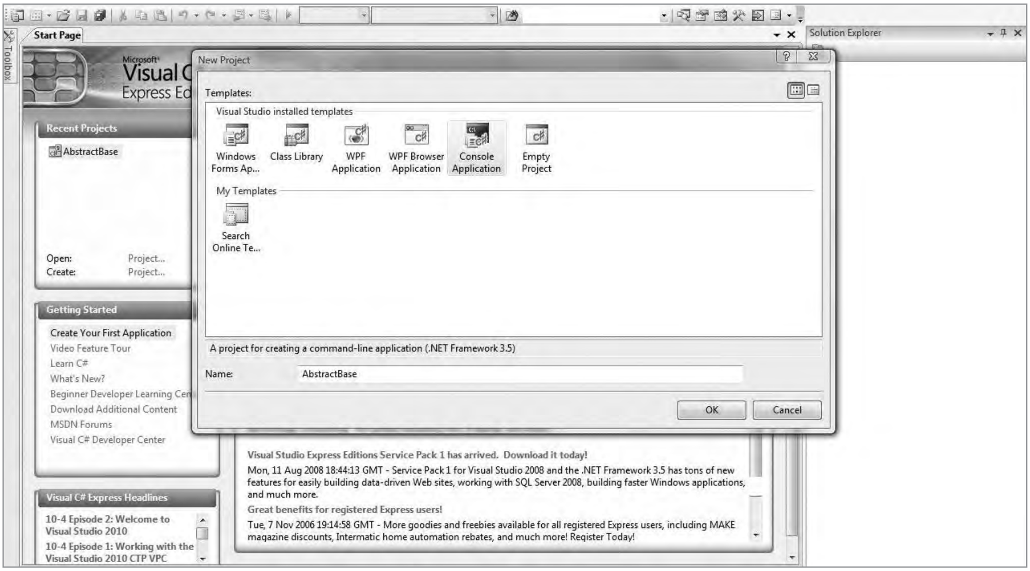


Figure 6-1. *Creating a C# console application*

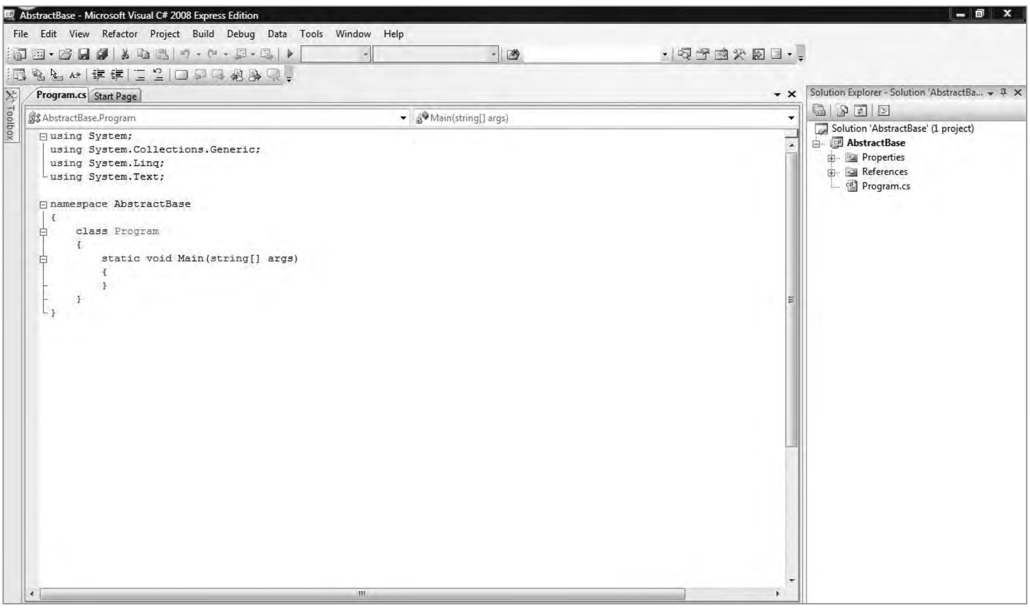


Figure 6-2. *Visual Studio created a basic structure for us.*

Even though Visual Studio was kind enough to create some code for us, we're going to start by creating a new file. Right-click on the *AbstractBase* project in the Solution Explorer, click Add, and then click Class. This will bring up a window for adding a new class file to the project. Call it *humanBeing.cs* and click Add (Figure 6-3).

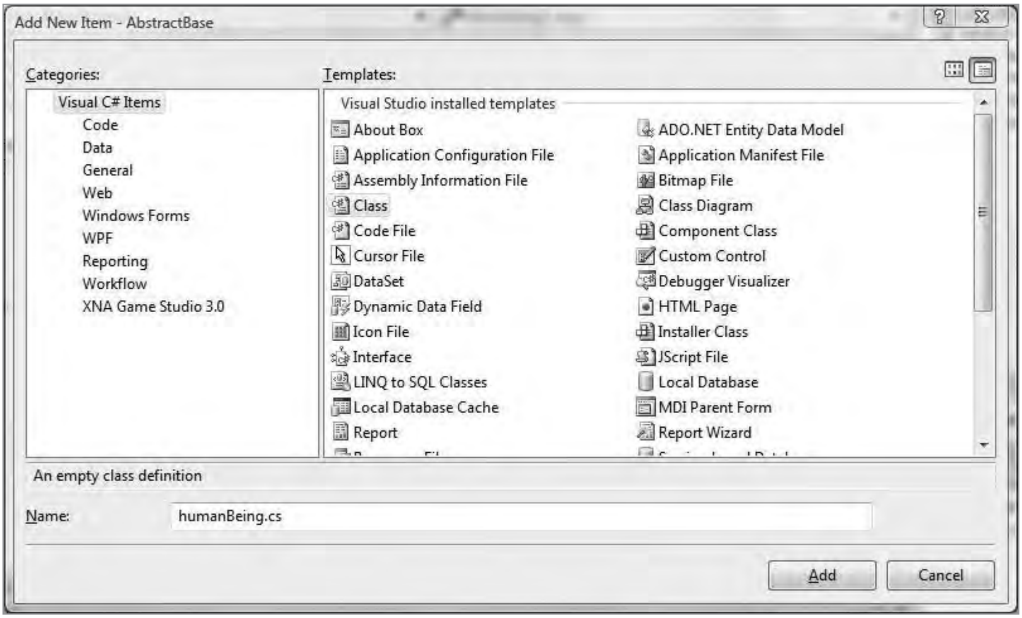


Figure 6-3. *Adding the humanBeing.cs file to the project*

As always, Visual Studio does a little legwork and provides some skeleton code in the file. You can clear that code entirely and replace it with Listing 6-1.

■ **Tip** C# is picky, picky, picky when it comes to syntax. This is not a bad thing; it's just a requirement that your language be very explicit. If you find errors after entering this code, make sure you've got all the semi-colons in the right place relative to the brackets, that the correct things are capitalized, and so on. What it does *not* care about is indentation, however.

Listing 6-1. *A Sample humanBeing.cs C# Abstract Base Class for a Human Being*

```
using System;

namespace AbstractBase
{
     abstract class humanBeing
     {
          public string name { get; private set; }
          public DateTime? dateOfBirth { get; private set; }

          public void SetName(string _name)
          {
               if (String.IsNullOrEmpty(_name)) { throw new Exception("Name cannot➥
be null."); }
               name = _name;
          }

        public void SetDateOfBirth(DateTime? _dateOfBirth)
        {
              if (!_dateOfBirth.HasValue) { throw new Exception("Date of birth➥
cannot be null."); }
              dateOfBirth = _dateOfBirth;
        }

       public abstract void Initialize();
       public abstract void CleanUp();
       public new abstract string ToString();
       }
}
```

That is a lot more typing than we're used to seeing. Indeed, as a statically typed language C# does require quite a bit of exposition to run properly. Some people swear by it, some absolutely hate it. What will be interesting to see in a few pages is how C#, with all its explicit typing, manages to interact pretty deftly with IronPython's freewheeling nature.

■ **Note** Why capitalize the project names when the variables and classes have a different style? It's really just what I'm comfortable with. The variables and classes use what's called *camel casing*, so named because the first letter is not capitalized, creating a word that looks like it has one or more humps. An example would be *calculateSalesTax*. I think project names that use this style look kind of funny, so I capitalize the first letter.

As far as putting things in context, the *humanBeing* base class we wrote defines a few properties for the entity: name and date of birth. We've set the *dateOfBirth* to be nullable so that we can check whether a valid value has been provided, a new feature in .NET. We marked the **mutator** as private so that the properties can be modified only by going through our public methods. If we allowed calling code to set the parameters directly, it prevents us from doing validity checks and other useful tasks.

■ **Note** Mutator is one-half of the geeky programming terms *mutator* and *accessor*, whose names are admittedly too cool for them. *Mutators* allow you to change the value of a property, and *accessors* allow you to retrieve those values. In our case, our accessors ("get") are public because we want code to access those variables easily, but the mutators ("set") are private, so calling code has to go through channels we've defined, allowing us a bit of control.

We defined a few abstract methods that handle the initialization and cleanup of the class. The last thing we did was to override the *.ToString()* method so that subclasses are required to implement their own version; we had to use the "new" keyword to identify that we want to override the basic version that .NET provides.

■ **Tip** If you take only one piece of development advice from this entire book, please take this one: *Always* include a *.ToString()* override method in your custom classes. Nothing is worse than using someone else's component, calling *.ToString()* on it, only to get something like "MyClass.MethodName" instead of useful output. Plus it can be very helpful for debugging purposes, as we'll see in a bit. The MSDN documentation has the following to say on the subject: "When you create a custom class or struct, you should override the *ToString* method in order to provide information about your type to client code." See `http://msdn.microsoft.com/en-us/library/ms173154(VS.80).aspx` for more details on this.

With a base class under our belt, we can create a class to implement that base class in a very similar fashion to what we did earlier in the book with IronPython. Add a new class

file called *person.cs* to the project and enter the code in Listing 6-2. Make sure to delete the code Visual Studio provided for you initially.

Listing 6-2. *A Sample person.cs C# Class That Implements the Human Being Base Class*

```
using System;
using System.Reflection;
using System.Text;

namespace AbstractBase
{
    class person : humanBeing
    {
        public override void Initialize()
        {
            SetName("Alan Harris");
            SetDateOfBirth(Convert.ToDateTime("1/1/2009"));
        }

        public override string ToString()
        {
            PropertyInfo[] p = GetType().GetProperties();
            StringBuilder sb = new StringBuilder();

            foreach (PropertyInfo pi in p)
            {
                sb.Append(pi.Name);
                sb.Append(": ");
                sb.Append(pi.GetValue(this, null));
                sb.Append("\r\n");
            }

            return sb.ToString();
        }

        public override void CleanUp()
        {
            // perform any cleanup tasks here...
        }
    }
}
```

The final step is actually to call an instance of the *person* class and see what output we get. We can modify the *Program.cs* file in the project to look like Listing 6-3. Once you have done so, press F5 to run the application (Figure 6-4).

Listing 6-3. *Using the Base Class in Program.cs*

```
using System;

namespace AbstractBase
{
    class Program
    {
        static void Main(string[] args)
        {
            person p = new person();
            p.Initialize();
            Console.WriteLine(p.ToString());
            Console.ReadLine();
            p.CleanUp();
        }
    }
}
```

■ **Tip** Note in Figure 6-4 that we changed the namespace of this file to *CSharpTestbed*, which matches the class files we'd created earlier. If you don't change the namespace, you'll find that C# complains about not being able to find the *person* class. If for whatever reason you don't want to change the namespace, you can add **using CSharpTestbed;** to the top of the *Program.cs* file and leave the namespace alone. Much like the IronPython import keyword, this tells C# to look for code in a specific place.

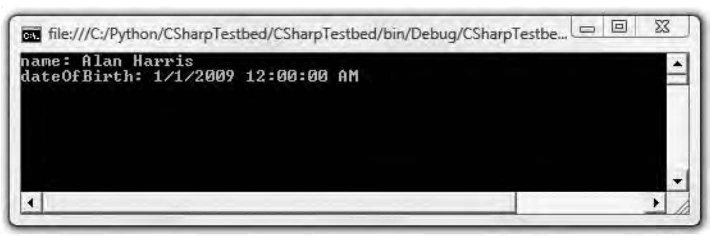


Figure 6-4. *Our calling class implements the base class.*

Don't worry about the syntax or implementation in C#. What's important here is the idea that the *person* object **can be substituted for the base class it implements**. The bit of code in Listing 6-4 is completely valid in C#. Make a new class file called *accepts.cs,* and add to it the code in Listing 6-4.

Listing 6-4. *A Demonstration of the Polymorphic Nature of the Base Class in accepts.cs*

```
using System;

class accepts
{
    public string AcceptsTypes(humanBeing h)
    {
        return h.name;
    }
}
```

This code expects an object of type *humanBeing* to be passed to it. What happens if we provide a person instead? Modify the *Program.cs* file to look like the code in Listing 6-5, and press F5 to try it out.

Listing 6-5. *Passing a Person to the accepts Class*

```
person p = new person();
p.Initialize();
Console.WriteLine(p.ToString());
p.CleanUp();

accepts a = new accepts();
Console.WriteLine(a.AcceptsTypes(p));
Console.ReadLine();
```

Although the *accepts* class is looking for an object of type *humanBeing*, we were able to substitute one of *person* type, which is a more specific subclass. Executing the code in Listing 6-5 presents the output in Figure 6-5.

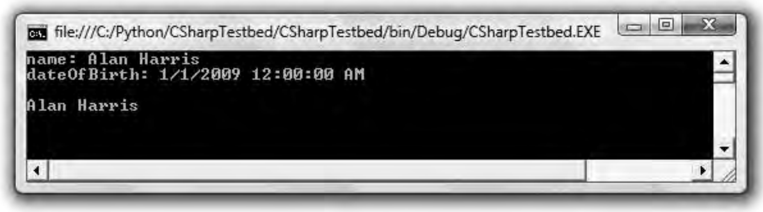


Figure 6-5. *The name is printed via the "a" object, which accepts humanBeing.*

Notice how much information we had to pass back and forth and how tightly that information is regulated and checked. It's all very explicit. IronPython takes a different approach. Let's look at a similar skeleton structure and see how IronPython handles it. You don't need to create a new project or the IronPython class files in Listing 6-6; they are simply illustrational with an end result that will be very similar to what you just did in C#. It is sufficient just to follow along and compare to the C# way of doing things.

Listing 6-6. *A Bare-Bones IronPython Base Class*

```
from System import *

class humanBeing(object):
    "Human being base class."

    def SetName(self, _name):
        self.__name = _name

    def GetName(self):
        return self.__name

    def SetDateOfBirth(self, _dateOfBirth):
        self.__dateOfBirth = _dateOfBirth

    def GetDateOfBirth(self):
        return self.__dateOfBirth

      def Initialize(self):
           pass

      def CleanUp(self):
           pass
```

```
    def ToString(self):
        pass

    name = property(GetName, SetName)
    dateOfBirth = property(GetDateOfBirth, SetDateOfBirth)
```

Now, you can substitute any class that inherits from *humanBeing* in place of *humanBeing* in any code that operates on it. First, let's look at a simple implementation of a person (Listing 6-7).

Listing 6-7. *An Implementation of a Person*

```
from System import *
from humanBeing import *

class person(humanBeing):
    "A person that inherits from humanBeing"

    def Initialize(self):
        SetName("Alan Harris")
        SetDateOfBirth(Convert.ToDateTime("1/1/2009"))

    def ToString(self, p):
        # perform any display tasks here
        pass

    def CleanUp(self):
        # perform any cleanup tasks here
pass
```

Listing 6-8 proves that we can substitute one class for another; so long as any methods called are implemented in both classes, the parent and child classes are totally interchangeable.

Listing 6-8. *The Polymorphic Nature of the Classes Allows Us to Substitute One for Another.*

```
p = person()
p.SetName('Alan Harris')
p.SetDateOfBirth(Convert.ToDateTime('1/1/2009'))
```

```
h = humanBeing()
h.SetName('Tom Smith')
h.SetDateOfBirth(Convert.ToDateTime('2/2/2009'))

objects = []
objects.append(p)
objects.append(h)

for o in objects: print o.GetName()
```

You could argue that our example is a bit contrived. Technically, any class that implemented a *.GetName()* method would work in Listing 6-8, and therein lies the entire point. IronPython relies extensively on what is called **duck typing**. It's a powerful feature, although mildly terrifying to developers used to statically typed languages.

■ **Note** *Duck typing*: If it looks like a duck and quacks like a duck, it must be a duck. This means that as a dynamic language IronPython will loosely allow one class to be substituted for another, enabling code to operate on differing object types and methods as though they were the same.

In IronPython, if two classes implement the same methods being used, then for all intents and purposes they are interchangeable. This presents some unique opportunities and really opens the development doors, although a careful eye is required to make sure things don't get out of control. Until you're used to that style of object handling, it's a little tricky to keep track of what's happening structurally behind the scenes, and you can encounter the occasional hard-to-squash bug. The flip side is that we're permitted to make some truly useful, reusable components without requiring our users to implement complex interfaces.

Plug and Play

In my opinion, one of the cooler uses for IronPython is as a **plug-in,** or scripting engine for applications written in other languages. As a C# developer by day, I find that using the IronPython libraries in my applications helps to lower the maintenance requirements of whatever project I'm currently working on—or allows rapid prototyping of new features. As such, I've come to rely on some simple design patterns that shorten even *those* tasks by quite a bit. The inheritance and polymorphism aspects of object-oriented programming are critical in this respect, as we'll quickly come to see.

■Note A *plug-in* is an extension to a parent application. For example, if you're a Firefox user, you're probably familiar with the various user-created add-ins that add or modify the functionality of the browser itself. I'm a big fan of the Web Developer Toolbar, Firebug, and ShowIP, among others. There are also plug-ins to switch between IE and Firefox tabs, which can save you a lot of time in web development.

Luckily, calling IronPython code from C# or VB is a straightforward task, but this has a few caveats. The relationship between IronPython classes and traditional .NET classes isn't 1-to-1. Consequently, you'll find one or two hoops to jump through. But if you create a small plug-in framework for reuse in other projects, even that workload can be significantly reduced.

The most notable hoop is the first one you'll hit: How exactly *do* you call IronPython from other .NET code? Microsoft has provided a *ScriptEngine* class that allows you to hook the IronPython engine in and communicate back and forth between your IronPython code and hosting code. Over the remainder of this chapter, we'll look at a way to set up the *ScriptEngine* in a reusable way; we don't want to have to create *ScriptEngines* all over the place. More to the point, we may want to reuse this plug-in architecture elsewhere. Reusability is going to be a running theme.

So where can you find this *ScriptEngine* class? When you install IronPython, several libraries are included for use in your .NET applications. The four we'll be using are **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Scripting.dll**, and **Microsoft.Scripting.Core.dll**. These expose a variety of functions that facilitate the use of IronPython as well as provide ways to modify and retrieve information from IronPython class files. Before we dig into a full plug-in architecture, we will build a simple application using those libraries and look at how to call our classes from C#.

Architecting Flexibility

Create a new Visual Studio project; select the Empty Project type, which will create an empty solution file for us. Call the project *Plugin*. Click OK to create the empty project solution (Figure 6-6).

■Tip For most projects, I find it best to create a parent solution that houses all subprojects. For example, if I were building an application for the accounting department, I could create an empty parent solution that has UI, Data, and Business projects underneath. This helps to enforce good separation of concerns and ideally promotes future code reuse.

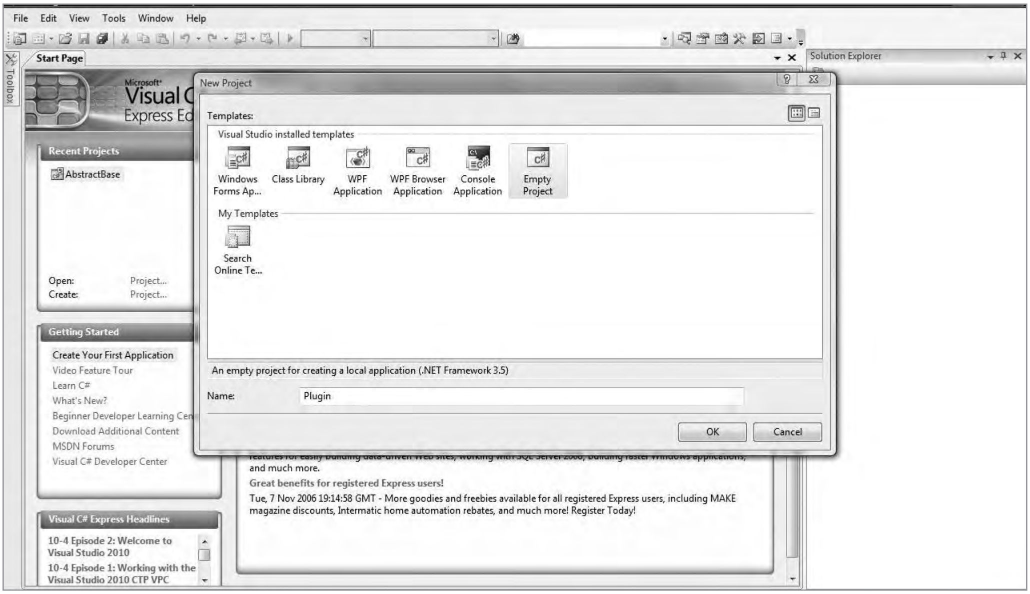


Figure 6-6. *Creating an empty parent solution called Plugin*

Next we'll add a C# project to our solution. Right-click on the empty solution and click Add ➤ New project. Add a new Console Application called *ConsoleUI* and click OK (Figure 6-7).

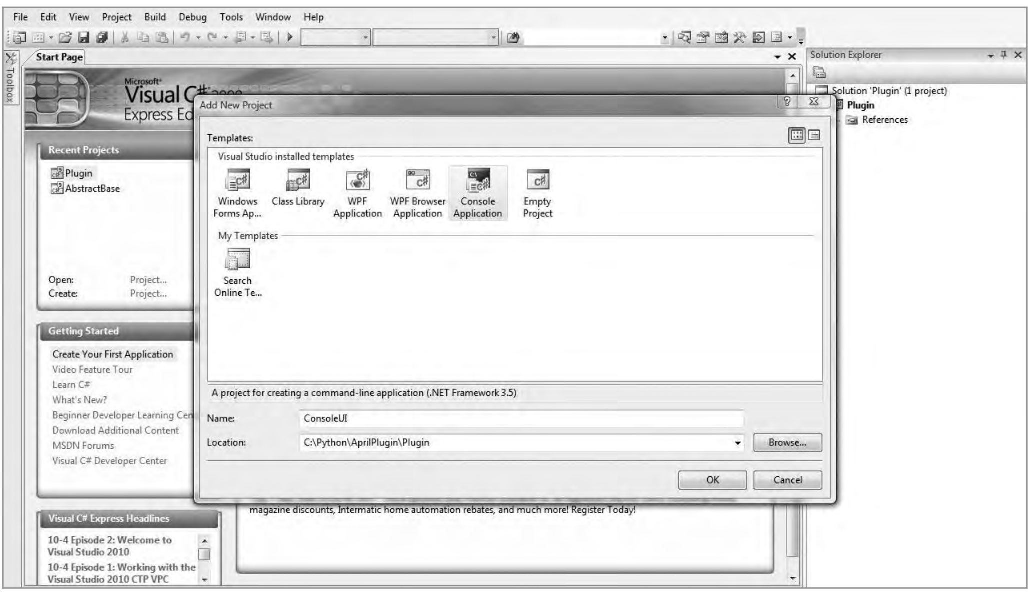


Figure 6-7. *Adding the ConsoleUI project to the Plugin solution*

Finally, we should add a class library to our project. Right-click on the solution and click Add ➤ New project. Add a new class library called *IPEngine* to the solution by clicking OK (Figure 6-8).

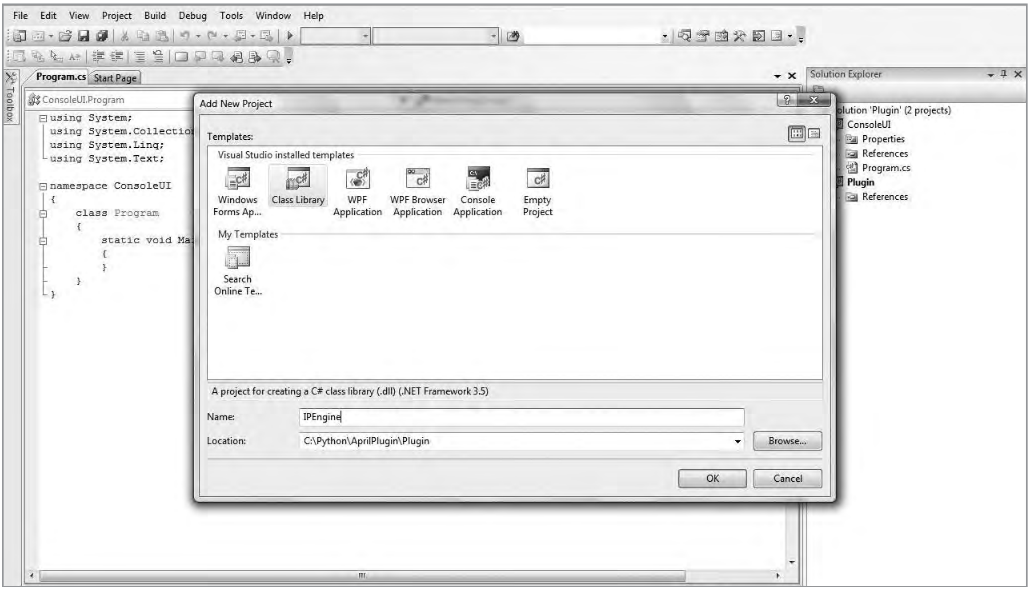


Figure 6-8. *Adding the IPEngine class library to the Plugin solution*

■ **Note** *Class libraries* are individual code components that can be added as references to other applications. They contain one or more classes and can expose methods and properties to calling code. They're extremely effective development tools for creating reusable code. The catch is that they are not directly executable like a typical application; their file extension is *.dll* (dynamically linked library), and they can only be used by a calling executable.

If your Visual Studio solution looks like Figure 6-9, then we're ready to get started!

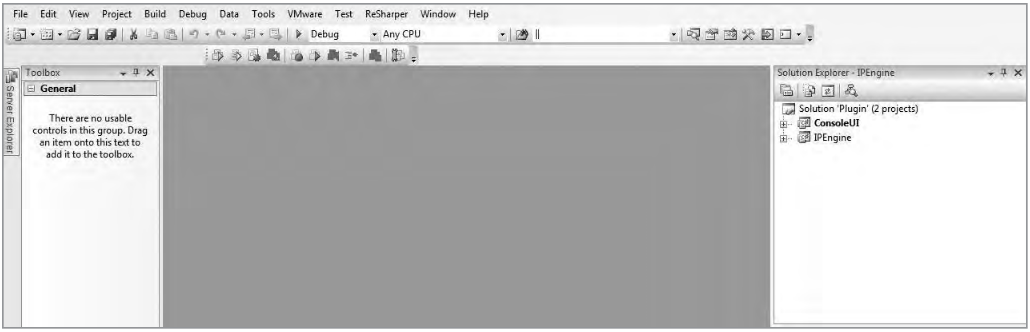


Figure 6-9. *The solution is set up.*

Calling IronPython Code

Before we can use IronPython code in our application, we need to add references to the libraries I mentioned earlier in the chapter. Right-click on the *IPEngine* project and click Add Reference. Click the Browse tab; you'll need to locate the directory on which IronPython is installed on your hard drive. For me, that's *C:\Program Files\IronPython 2.0*. Once you've browsed to the correct folder, you'll see a variety of files listed in the folder. Hold down the Ctrl key and left-click on **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Scripting.dll**, and **Microsoft.Scripting.Core.dll**. Click OK to add them to the *IPEngine* project (Figure 6-10).

■ **Caution** Unlike some of our earlier applications, this one is of an increased difficulty and requires a bit more coding on our part. As a result, compiling it at various early stages will produce errors. *This is totally fine*. The application has a lot of wiring in it and I'm covering things in a specific order. If things aren't compiling but you haven't finished the section yet, it's likely that something's about to be added but needs explanation to put it in context first.

The *IPEngine* project will contain a class file called *Class1.cs*. Rename that to *EngineManager.cs*. At the top of the file, we need to import the code from our libraries that we want to use in our file (Listing 6-9).

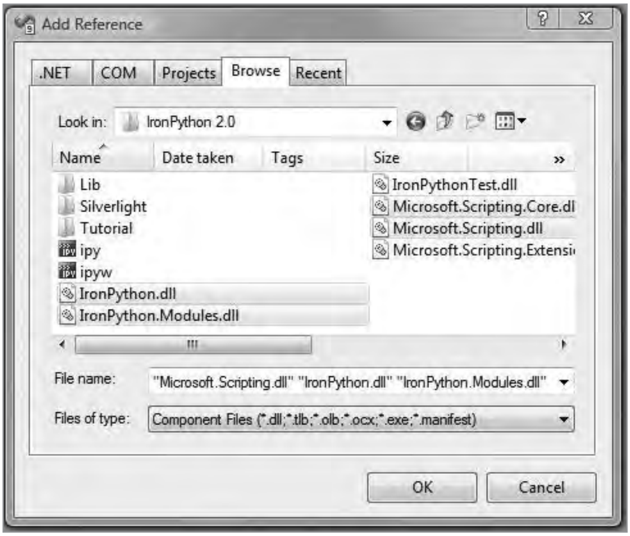


Figure 6-10. *We need to add the IronPython libraries to our project.*

Listing 6-9. *The Beginning of the EngineManager Class*

```
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;

namespace IPEngine
{
    public class EngineManager
    {
    }
}
```

Our communication with IronPython code will be handled by an object called the *ScriptEngine*. It relies on *ScriptSource, ScriptScope,* and *ObjectOperations* objects to pass data between IronPython and calling classes. Let's add private variables to the class that set up each of these (Listing 6-10).

Listing 6-10. *Setting Up the Necessary Objects for IronPython Communication in EngineManager.cs*

```
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;

namespace IPEngine
{
    public class EngineManager
    {
        private ScriptEngine engine;
        private ScriptSource source;
        private ScriptScope scope;
        private ObjectOperations operations;
    }
}
```

Remember that we always want to hide as much information as possible behind abstractions. Therefore, let's create an *Initialize* method to handle the setup workload for us so that the details are invisible to calling code (Listing 6-11).

Listing 6-11. *Creating an Initialize Method to Handle Setup Within EngineManager.cs*

```
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;

namespace IPEngine
{
    public class EngineManager
    {
        private ScriptEngine engine;
        private ScriptSource source;
        private ScriptScope scope;
        private ObjectOperations operations;

        /// <summary>
        /// Sets up the IPEngine for use by calling code.
        /// </summary>
        /// <param name="file">The name of the IronPython source file to➥
execute.</param>
```

```
        public void Initialize(string file)
        {
            engine = Python.CreateEngine();
            source = engine.CreateScriptSourceFromFile(file);
            scope = engine.CreateScope();
            operations = engine.Operations;
        }
    }
}
```

■**Note** The `///` comments at the top of the method are XML documentation. This is similar to the use of *docstring* in Python, in that it describes the method for use in helpful instructions elsewhere. They are totally optional, but I find them to be quite useful. When you call these methods in the *ConsoleUI* application later, you'll see this information appear in the IntelliSense that shows up as you type, so a mix of clarity and brevity is best. In general, "what" and "why" make for better comments than "how"; I'll read your code to see how you accomplished something; I'll read your comments if I need to figure out why.

Next we need to provide a mechanism by which a class can be referenced and a method called. We'll create an *Execute* method to handle this (Listing 6-12).

Listing 6-12. *The First Version of the EngineManager Class Completed*

```
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;

namespace IPEngine
{
    public class EngineManager
    {
        private ScriptEngine engine;
        private ScriptSource source;
        private ScriptScope scope;
        private ObjectOperations operations;

        /// <summary>
        /// Sets up the IPEngine for use by calling code.
        /// </summary>
        /// <param name="file">The name of the IronPython source file to➥
execute.</param>
```

```
        public void Initialize(string file)
        {
            engine = Python.CreateEngine();
            source = engine.CreateScriptSourceFromFile(file);
            scope = engine.CreateScope();
            operations = engine.Operations;
        }

        /// <summary>
        /// Gets the results of an IronPython file after execution.
        /// </summary>
        /// <typeparam name="EngineResults">Generic result from IronPython➥
class.</typeparam>
        /// <param name="className">The name of the class to reference.</param>
        /// <param name="methodName">The name of the method to execute.</param>
        /// <returns>The results of IronPython execution.</returns>
        public EngineResults Execute<EngineResults>(string className,➥
string methodName)
        {
            source.Execute(scope);
            var classObj = scope.GetVariable(className);
            var classInstance = operations.Call(classObj);
            var classMethod = operations.GetMember(classInstance, methodName);
            var results = (EngineResults) operations.Call(classMethod);
            return results;
        }
    }
}
```

Tip The *Execute* method has some unique syntax to it. The `<EngineResults>` bit implies that we are returning some generic information back to the calling class. We're telling C# that we don't know if it's going to be a string, an int, or some type of object. It will be up to the calling code to sort out those details.

That's it—seriously. We can use this simple class to execute any IronPython code we want. But we should start small; the more complexity we introduce, the greater the chance for things to get messy. You'll have to go outside Visual Studio for this next step: navigate to the folder on your drive where the *Plugin* solution lives and create a new folder in it called *Scripts*. Now open your text editor of choice and create in that folder a

file called *pluginTest.py*. We'll house our IronPython scripts in this folder to keep things nice and neat.

In the *pluginTest* file, enter the code shown in Listing 6-13 and save it.

Listing 6-13. *A Very Basic IronPython Class for Testing the Plug-in Library*

```
class pluginTest:
    def HelloPlugin(self):
        message = 'Hello via the plugin!'
        return message
```

■ **Note** This whole file and folder creation is expedited significantly in the commercial version of Visual Studio, where you can add "Solution Folders" and so on directly in the IDE.

Now select Build ➤ Build Solution and correct any errors that appear (hopefully, none!). Once the build is successful, right-click on the *ConsoleUI* project and select Add reference. Navigate to the folder that contains the *IPEngine* project; inside you'll see several folders. You'll want to navigate to *bin* and then to *Debug*. In the *Debug* folder you'll see several *dlls*. Select *IPEngine.dll* and click OK to add it to the *ConsoleUI* project.

Having added it, let's try it out. Open the *Program.cs* file; modify it to look like the code in Listing 6-14. You'll need to modify the path for the IronPython script file to match where you've created the project.

Listing 6-14. *Calling the Plug-in Library to Run Our pluginTest Code*

```
using System;
using IPEngine;

namespace ConsoleUI
{
    class Program
    {
        static void Main(string[] args)
        {
            var e = new EngineManager();
            e.Initialize(@"C:\Python\Plugin\Scripts\pluginTest.py");
            Console.WriteLine(e.Execute<string>("pluginTest", "HelloPlugin"));
            Console.Read();
        }
    }
```

■ **Tip** Here's the follow-up to the generic method defined in the library. Our calling code tells the library that we expect the return type of the *Execute* method to be of type *string*. This is a classic example of the differences between static and dynamically typed languages: in pure Python we wouldn't need to be so very explicit, nor would we make accommodations specifically with the intention of allowing generic access.

Now, if you run this application by pressing F5, you should find that your IronPython script is executed (Figure 6-11).



Figure 6-11. *IronPython via C#*

■ **Tip** Frequently in solutions where you have class libraries (particularly multiple class libraries), one library may be dependent on another one in your solution. If the build order is not correct, you'll continually get errors and it may not be clear why. Right-click on the solution and select Project Build Order. Here you can adjust the order in which projects are compiled when you build or run the solution; in our case, make sure *IPEngine* is listed before *ConsoleUI*. You can also set dependencies in the aptly named tab; *ConsoleUI* depends on *IPEngine*. In general, if you've built with an *n*-tier architecture in mind, the UI will be the last thing to get compiled and will depend directly on any business-tier libraries, which themselves will depend on data-tier libraries.

Creating a Plug-in Base

Although what we have is quite successful in terms of executing our IronPython code, it would benefit us as developers to standardize our plug-ins with a base class from which to inherit so that any plug-in object we want to use is guaranteed to have some basic features.

A useful base class for a plug-in would have the following properties:

- A unique ID for the plug-in instance
- A useful display name
- The location of the plug-in on disk
- An enumeration that describes the current status of the plug-in
- The name of the class to use
- The name of the method to execute

In terms of our enumeration for plug-in status, we can start with the following:

1. Unavailable
2. Loaded
3. Failed

Right click on the *IPEngine* project and click Add New Item. Add to the project a class called *BasePlugin.cs*. Modify the code in *BasePlugin* to look like the code in Listing 6-15.

Listing 6-15. *The Abstract Base Class for Our Plug-ins in BasePlugin.cs*

```
using System;

namespace IPEngine
{
    public abstract class BasePlugin
    {
        public enum pluginStatus
        {
            Unavailable,
            Loaded,
            Failed
        }

      public string id { get; private set; }
        public string displayName { get; private set; }
        public string className { get; private set; }
        public string methodName { get; private set; }
        public string fileLocation { get; private set; }
        public pluginStatus status { get; private set; }
```

```
    /// <summary>
        /// Initializes a unique ID for a plugin.
        /// </summary>
        public void SetPluginID()
        {
            if (!String.IsNullOrEmpty(id)) { throw new Exception("Plugin id➥
has already been defined."); }
            id = Guid.NewGuid().ToString().ToLower();
        }

        /// <summary>
        /// Sets the friendly display name for a plugin.
        /// </summary>
        /// <param name="_displayName">The string to display.</param>
        public void SetDisplayName(string _displayName)
        {
            if (String.IsNullOrEmpty(_displayName)) { throw➥
new Exception("Display name cannot be null."); }
            displayName = _displayName;
        }

        /// <summary>
        /// Sets the name of the IronPython class to use.
        /// </summary>
        /// <param name="_className">The name of the class.</param>
        public void SetClassName(string _className)
        {
            if (String.IsNullOrEmpty(_className)) { throw➥
new Exception("Class name cannot be null."); }
            className = _className;
        }

        /// <summary>
        /// Sets the name of the IronPython method to execute.
        /// </summary>
        /// <param name="_methodName">The name of the method.</param>
        public void SetMethodName(string _methodName)
        {
            if (String.IsNullOrEmpty(_methodName)) { throw➥
new Exception("Method name cannot be null."); }
            methodName = _methodName;
        }
```

```
        /// <summary>
        /// Sets the location of the plugin on disk.
        /// </summary>
        /// <param name="_fileLocation">The path of the file on disk.</param>
        public void SetFileLocation(string _fileLocation)
        {
            if (String.IsNullOrEmpty(_fileLocation)) { throw➥
new Exception("File location cannot be null."); }
            fileLocation = _fileLocation;
        }

        /// <summary>
        /// Sets the current status of the plugin.
        /// </summary>
        /// <param name="_status">The status as defined by the pluginStatus➥
enumeration.</param>
        public void SetStatus(pluginStatus _status)
        {
            if (status == _status) return;
            status = _status;
        }

        // classes that inherit from this base must implement the following methods
        public abstract void ConfigurePlugin(string _displayName,➥
string _className, string _methodName,  string _fileLocation);
        public abstract ExecuteResults ExecutePlugin<ExecuteResults>();
        public abstract new string ToString();
    }
}
```

Although Listing 6-15 looks a bit long, it actually serves a few very straightforward purposes. Its primary functions are to define the status types for a plug-in and to handle setting the properties of the plug-in. Classes that inherit from this base class have access to all these properties and the enumeration defined within. It also defines two methods, *ConfigurePlugin* and *ExecutePlugin*, that inheriting classes have to implement. Let's create in the IPEngine project a new class file called *TestPlugin.cs* (Listing 6-16).

Listing 6-16. *A Test Plug-in That Needs an ExecutePlugin Method Implemented*

```
using System;
using System.Text;

namespace IPEngine
{
    public class TestPlugin : BasePlugin
    {
        private EngineManager em;

       public override void ConfigurePlugin(string _displayName,➥
string _className, string _methodName, string _fileLocation)
        {
            try
            {
                // set up the plugin properties
                SetPluginID();
                SetDisplayName(_displayName);
                SetClassName(_className);
                SetMethodName(_methodName);
                SetFileLocation(_fileLocation);
                SetStatus(pluginStatus.Loaded);

      // set up the instance of the plugin engine
                em = new EngineManager();
                em.Initialize(fileLocation);
            }
            catch
            {
                SetStatus(pluginStatus.Failed);
            }
        }

      public override ExecuteResults ExecutePlugin<ExecuteResults>()
        {
             // to do: implement this method
            throw new Exception("I have to do something to compile…");

        }
```

```
        public override string ToString()
        {
            var s = new StringBuilder();
            s.Append(displayName);
            s.Append(": ");
            s.Append(className);
            s.Append(" calling method ");
            s.Append(methodName);
            return s.ToString();
        }
    }
}
```

■ **Note** Please note that I practice what I preach! We have overridden a *ToString* method here that displays a string in the form "*display name* : *class* calling method *method*." Again, it's not a requirement, but it does make debugging easier. Now if I need to do a quick sanity check on the state of the object, I can call *pluginName*.ToString() and get a quick view at the object's contents.

The reason we haven't implemented the *ExecutePlugin* method yet is simple: we don't have anything in our *EngineManager* to handle our plug-in! Let's remedy that. Open the *EngineManager* class and add below the existing *Execute* method shown in Listing 6-17.

Listing 6-17. *Overriding the Execute Method in EngineManager to Accommodate Plug-ins*

```
/// <summary>
/// Gets the results of an IronPython plugin after execution.
/// </summary>
/// <typeparam name="EngineResults">Generic result from➥
IronPython class.</typeparam>
/// <param name="plugin">A plugin that implements BasePlugin.</param>
/// <returns>The results of IronPython execution.</returns>
public EngineResults Execute<EngineResults>(BasePlugin plugin)
```

```
{
    source.Execute(scope);
    var classObj = scope.GetVariable(plugin.className);
    var classInstance = operations.Call(classObj);
    var classMethod = operations.GetMember(classInstance, plugin.methodName);
    var results = (EngineResults)operations.Call(classMethod);
    return results;
}
```

It's pretty similar to the existing *Execute* method. The only real differences are that it accepts any object that inherits from *BasePlugin* and that it uses properties of that plug-in (such as *plugin.className*) instead of specific strings. Now let's jump back to our *TestPlugin* class and implement that *ExecutePlugin* method we left hanging (Listing 6-18).

Listing 6-18. *Overriding the Execute Method in EngineManager to Accommodate Plug-ins*

```
public override ExecuteResults ExecutePlugin<ExecuteResults>()
{
    switch (status)
    {
        case pluginStatus.Loaded:
            return em.Execute<ExecuteResults>(this);

        case pluginStatus.Failed:
            throw new Exception("EngineManager failed to initialize plugin.");

        case pluginStatus.Unavailable:
            throw new Exception("Plugin is unavailable; please initialize➥
properly.");

        default:
            throw new Exception("Unable to verify plugin status; please➥
initialize properly.");
     }
}
```

This *ExecutePlugin* method now evaluates the current value of the *pluginStatus* property to find out whether to execute the IronPython method or throw an exception back to the calling class to indicate some type of failure has occurred. Now that we have a more robust plug-in system, let's modify the main UI program to use it (Listing 6-19), and we'll compare the direct *EngineManager* to the more abstract plug-in class and see how the terrain looks in a new light (Figure 6-12).

Listing 6-19. *A Modified Program.cs in the ConsoleUI Application*

```
using System;
using IPEngine;

namespace ConsoleUI
{
    class Program
    {
        static void Main(string[] args)
        {
            var p = new TestPlugin();
            p.ConfigurePlugin("test", "pluginTest", "HelloPlugin",➥
@"C:\Python\Plugin\Scripts\pluginTest.py");
            Console.WriteLine(p.ExecutePlugin<string>());

            var e = new EngineManager();
            e.Initialize(@"C:\Python\Plugin\Scripts\pluginTest.py");
            Console.WriteLine(e.Execute<string>("pluginTest", "HelloPlugin"));
            Console.Read();
        }
    }
}
```

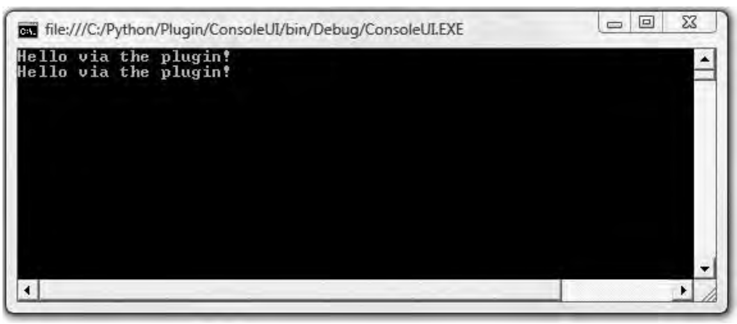


Figure 6-12. *The basic plug-in architecture is working.*

Choices, Choices

Is it better to use a direct call to the *EngineManager* or better to go through the plug-in architecture? I would recommend using the plug-in architecture; although both classes provide a standardized way to access IronPython code, you can easily extend the

BasePlugin class to other types, and my personal experience is that a layer of abstraction generally fixes everything. As I've mentioned on a few occasions throughout this book, you're doing yourself a service if you introduce a little flexibility as you go rather than trying to tack something on later. Don't take my word though; there are no hard-and-fast rules. For some applications, developing a plug-in system like the one we've created so far is, frankly, overkill. For other applications, we would need to take this system and expand on it greatly. It's all relative.

There's another, more subtle benefit happening here. The user interface code has no clue what's happening behind the scenes. All it knows is that there's a *TestPlugin* class with a handful of methods; it doesn't know anything about the IronPython *ScriptEngine* or any of that business. That's the way things should be. Granted, the *EngineManager* hides those specific implementation details too, but it still has the slight scent of "low-level" code. Code with that aroma seems like it should be tucked in the background somewhere, used when needed, and otherwise not seen. For that reason alone, my vote's on the plug-in architecture. You can always make calls to the *EngineManager* directly if you need to.

Supporting Healthy Arguments

The last expansion to the plug-in system will be an overloaded method that supports the passing of one or more arguments to an IronPython class. First, open the *BasePlugin.cs* file and add immediately after the existing *ExecutePlugin* method the overload method shown in Listing 6-20.

Listing 6-20. *Overloading ExecutePlugin to Accept Parameters*

```
public abstract ExecuteResults ExecutePlugin<ExecuteResults>(string[] parameters);
```

Next we need to overload the *ExecutePlugin* method in the *TestPlugin* class so that our parameters can be passed to the *EngineManager* properly. Add after the existing *ExecutePlugin* method in *TestPlugin.cs* the code shown in Listing 6-21.

Listing 6-21. *Overloading ExecutePlugin in the TestPlugin to Accept Parameters*

```
public override ExecuteResults ExecutePlugin<ExecuteResults>(string[] parameters)
        {
            switch (status)
            {
                case pluginStatus.Loaded:
                    return em.Execute<ExecuteResults>(this, parameters);
```

```
                case pluginStatus.Failed:
                    throw new Exception("EngineManager failed to initialize➥
plugin.");

                case pluginStatus.Unavailable:
                    throw new Exception("Plugin is unavailable; please initialize➥
properly.");

                default:
                    throw new Exception("Unable to verify plugin status; please➥
initialize properly.");
            }
        }
```

The method signature now indicates that the method is willing to accept an alternate call that accepts an array of strings as parameters to use in the method. Compare the code in Listing 6-21 to the code in Listing 6-18 and you can see how the method has been overridden to accept an array of strings. Note that we're also passing these parameters down to the *EngineManager.Execute* method, which as of now does not accept an array of parameters. To fix this, we'll overload the *Execute* method in the *EngineManager* class to accept parameters as well. Place the code in Listing 6-22 immediately after the existing *Execute* method.

Listing 6-22. *Overloading Execute to Accept Parameters in EngineManager.css*

```
/// <summary>
/// Gets the results of an IronPython plugin after execution.
/// </summary>
/// <typeparam name="EngineResults">Generic result from➥
IronPython class.</typeparam>
/// <param name="plugin">A plugin that implements BasePlugin.</param>
/// <param name="parameters">An array of string parameters.</param>
/// <returns>The results of IronPython execution.</returns>
public EngineResults Execute<EngineResults>(BasePlugin plugin, string[] parameters)
{
     source.Execute(scope);
     var classObj = scope.GetVariable(plugin.className);
     var classInstance = operations.Call(classObj);
     var classMethod = operations.GetMember(classInstance, plugin.methodName);
     var results = (EngineResults)operations.Call(classMethod, parameters);
     return results;
}
```

Now let's write a little IronPython code that tests this out. Add a new script to the *Scripts* folder called *pluginParameters.py* (Listing 6-23).

Listing 6-23. *pluginParameters.py, A Quick IronPython Script to Test Our Parameter Methods*

```
class pluginParameters:
      def tryParams(self, name, age):
          return "Hello, " + name + ", you are " + age + " years old."
```

The last step is to try calling our overloaded plug-in methods to see if everything works as intended. In the *ConsoleUI* project, open the *Program.cs* file and modify it to look like Listing 6-24; then run the solution (Figure 6-13).

Listing 6-24. *A Test of the New Plug-in Overloads in Program.cs*

```
using System;
using IPEngine;

namespace ConsoleUI
{
    class Program
    {
        static void Main(string[] args)
        {
            var t = new TestPlugin();
            t.ConfigurePlugin("test", "pluginTest", "HelloPlugin",➥
@"C:\Python\Plugin\Scripts\pluginTest.py");
            Console.WriteLine(t.ExecutePlugin<string>());

            var p = new TestPlugin();
            p.ConfigurePlugin("params", "pluginParameters", "tryParams",➥
@"C:\Python\Plugin\Scripts\pluginParameters.py");
            string[] parameters = {"Alan", "24"};
            Console.WriteLine(p.ExecutePlugin<string>(parameters));
```

```
            var e = new EngineManager();
            e.Initialize(@"C:\Python\Plugin\Scripts\pluginTest.py");
            Console.WriteLine(e.Execute<string>("pluginTest", "HelloPlugin"));
            Console.Read();
        }
    }
}
```



Figure 6-13. *The plug-in overloads now accept an array of parameters.*

"Somebody's Watching Me"

We should put our plug-in system to the test in a real-life example. Much like *NotQuitePad* in Chapter 5, we'll lay out a set of design requirements, plan the application, and implement it step by step. For this application, what we're going to build is a small Forms application that uses plug-ins to watch the file system and take various actions depending on what sort of activities have occurred. We'll use a mix of C# and IronPython: the C# side will be the form itself, and the IronPython side will handle all of the application's desired file system activities.

The Plan

As always, it's a good idea to start off with a basic plan so that we know where we're heading. We'll leave a little wiggle room for future development as well as for unforeseen problems or changes along the way.

1. The application should consist of a single form.

2. The application should monitor the file system for various changes.

3. The application should handle all actions based on file system changes via IronPython plug-ins.

The Design

What I envision for this application is a single form that has a text box control on it as well as two check box controls. The purpose of the text box is to display to the user any output or feedback based on file system changes; IronPython code will provide that feedback. The check boxes will activate or deactivate individual plug-ins. We'll test this at each step of the way by performing the task a given plug-in should be watching for. If the given check box is checked, we should see the task performed; otherwise it should be ignored.

First we'll create a new project in the *Plugin* solution. We don't necessarily have to do it in this solution; it's just convenient for the moment in case we want to debug or modify code in the *IPEngine* library. Right-click on the solution and click Add ➤ New Project. Select Visual C# ➤ Windows ➤ Windows Forms Application, and name it *FSWatcher*. Click OK to add it to the solution.

Next, right-click on the *FSWatcher* project and click Add Reference. Add to the project a reference to *IPEngine.dll* and click OK. Now rebuild the entire solution by clicking Build ➤ Rebuild Solution from the menu bar; everything should rebuild with no errors or warnings.

To accomplish the task of monitoring the file system, we're going to use an object .NET provides called the **FileSystemWatcher**. This object allows us to monitor a variety of actions in the file system, such as adding or deleting a file and watching for only specific types of files. It's actually quite a powerful tool. Our program will monitor *C:\Python* for the presence of *.p1* and *.p2* files. The tasks will be as follows:

1. **.p1**—inform the calling application that the file has been deleted from the folder
2. **.p2**—inform the calling application that the file has been added to the folder

---

■ **Note** You can find more information on the *FileSystemWatcher* at `http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.aspx`.

Writing the Basic IronPython Classes

For now, we'll keep the IronPython classes simple. I find development tasks are easier when I start with a simple, easily tested foundation. The first class we'll create in the *Scripts* folder is called *p1Handler.py* (Listing 6-25).

Listing 6-25. *p1Handler Will Return Information to the Calling Application*

```
class p1Handler:
      "Informs the calling application that a file is deleted."
      def NotifyCaller(self, fileName):
          return fileName + " was deleted."
```

This first handler informs the calling application that a file has been deleted from the folder we're watching. The next handler, *p2Handler.py*, will inform the calling application that a file has been added to the folder we're watching (Listing 6-26).

Listing 6-26. *p2Handler Will Also Return Information to the Calling Application*

```
class p2Handler:
      "Informs the calling application that a file has been added."
      def NotifyCaller(self, fileName):
          return fileName + " was added."
```

Creating the Parent Application

In the *FSWatcher* application, we have to set up the main form with the controls we need and make sure everything is named something useful. First, open the form in Design View and double-click on *GroupBox* in the toolbox. Resize it until it's nicely centered in the form, and change the *Text* property to *File System Watcher* (Figure 6-14).



Figure 6-14. *The GroupBox is a nice container for our controls.*

Next we'll add the text box control to the GroupBox. Double-click on the *TextBox* control in the toolbox and set the *Multiline* property to True, *Scrollbars* to Vertical, and the name to *txtUpdates*. Resize it to approximately two-thirds the size of the GroupBox. You should also set the *ReadOnly* property to True; this will prevent users from typing in the updates box and disrupting the results (Figure 6-15).

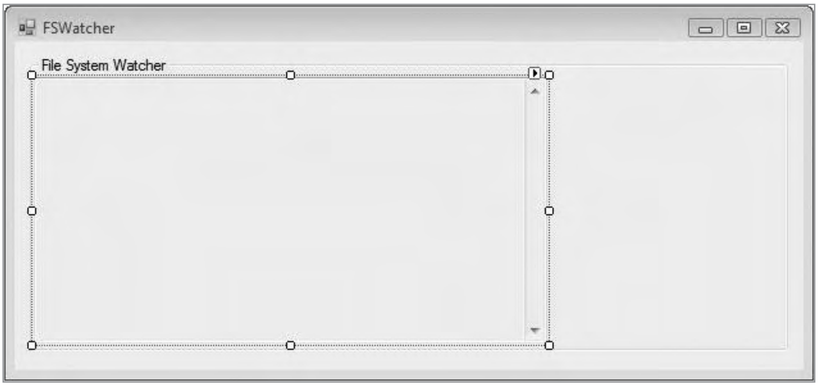


Figure 6-15. *The updates text box once all the properties have been set*

Now we need to add two check box controls off to the right. Name them *p1WatcherEnabled* and *p2WatcherEnabled*, respectively. Set the *Text* properties to *Watch .p1 files* and *Watch .p2 files* (Figure 6-16).

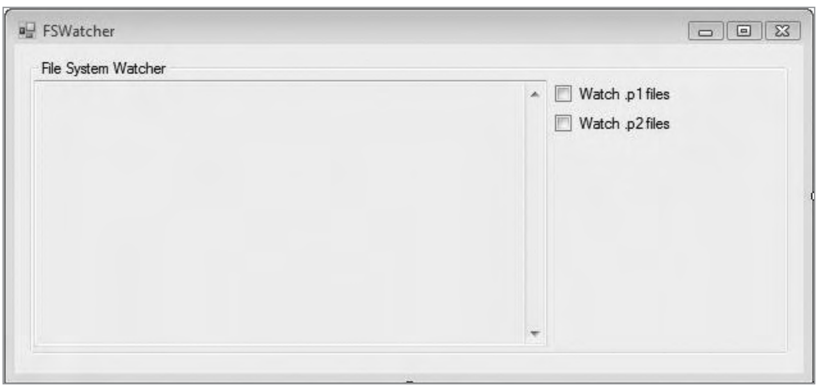


Figure 6-16. *The FSWatcher UI is complete.*

Wiring Things Together

With the UI set up, we can start wiring our code together. The first thing we'll do is run a quick test to make sure that our *IPEngine* reference is correct and that things are generally the way we expect them to be.

Open *Form1.cs* and modify the code to look like Listing 6-27. Then we'll run it (Figure 6-17).

Listing 6-27. *The Main Form with a Basic Plug-in Test Added*

```
using System;
using System.Windows.Forms;
using IPEngine;

namespace FSWatcher
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            var p = new TestPlugin();
            p.ConfigurePlugin("test", "p1Handler", "NotifyCaller",➥
@"C:\Python\Plugin\Scripts\p1Handler.py");
            string[] parameters = {@"C:\Python\test.p1"};
            txtUpdates.Text = p.ExecutePlugin<string>(parameters);
        }
    }
}
```

■ **Note** I created a dummy file called *test.p1* in my *C:\Python* folder. It doesn't matter what's in there—this is just a test at this point.

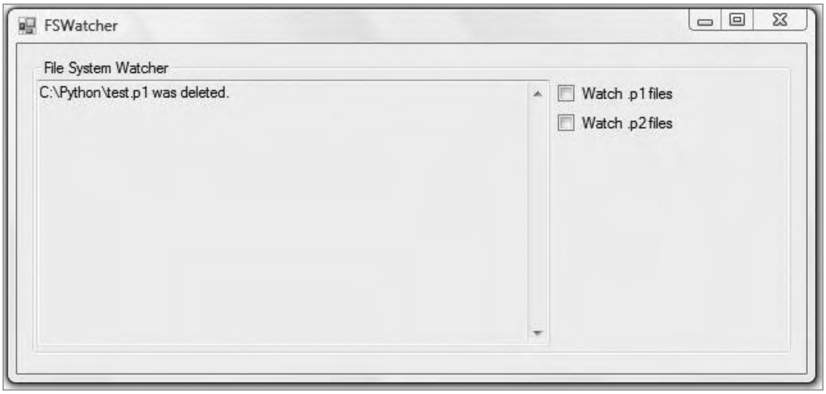


Figure 6-17. *A first run of the UI shows that our IronPython class is called.*

With the knowledge that our plug-in architecture is working, let's take a look at the *FileSystemWatcher* object and see how it can help us out.

First, on the main form, add a new *Timer* control and set the *Interval* property to 250 (which is expressed in milliseconds.) We'll use the Timer to fire every 250ms and update the *txtUpdates* control with anything that has happened; make sure that *Timer property Enabled* is set to True.

With the Timer in place, open the *Form1* code and modify it to look like Listing 6-28. We'll discuss it afterwards.

Listing 6-28. *The Main Form with FileSystemWatcher Added*

```
using System;
using System.IO;
using System.Windows.Forms;
using IPEngine;

namespace FSWatcher
{
    public partial class Form1 : Form
    {
        private static string updateText = String.Empty;
        private FileSystemWatcher fp1;
        private FileSystemWatcher fp2;
```

```
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            fp1 = InitializeP1Watcher(@"C:\Python", "*.p1");
            fp2 = InitializeP2Watcher(@"C:\Python", "*.p2");
        }

        public FileSystemWatcher InitializeP1Watcher(string path, string filter)
        {
            var f = new FileSystemWatcher { Path = path, Filter = filter,➥
EnableRaisingEvents = true };
            f.Deleted += OnP1Deleted;
            return f;
        }

       public FileSystemWatcher InitializeP2Watcher(string path, string filter)
        {
            var f = new FileSystemWatcher { Path = path, Filter = filter,➥
EnableRaisingEvents = true };
            f.Created += OnP2Created;
            return f;
        }

        private static void OnP1Deleted(object sender, FileSystemEventArgs e)
        {
            var p = new TestPlugin();
            p.ConfigurePlugin("test", "p1Handler", "NotifyCaller",➥
@"C:\Python\Plugin\Scripts\p1Handler.py");
            string[] parameters = { e.FullPath };
            updateText += p.ExecutePlugin<string>(parameters);
            updateText += "\r\n";
        }
```

```
        private static void OnP2Created(object sender, FileSystemEventArgs e)
        {
            var p = new TestPlugin();
            p.ConfigurePlugin("test", "p1Handler", "NotifyCaller",➥
@"C:\Python\Plugin\Scripts\p2Handler.py");
            string[] parameters = { e.FullPath };
            updateText += p.ExecutePlugin<string>(parameters);
            updateText += "\r\n";
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            txtUpdates.Text = updateText;
        }
    }
}
```

■ **Note** I really, really do not like the *Timer* control. It does have its uses, but I try to avoid it. I'm using it here to keep things simple. The point is the use of IronPython code via C#, not how to wire a variety of *EventHandlers* together.

Once we've created the form, we set up two *FileSystemWatcher* objects. One of them monitors the file system for the deletion of *.p1* files, and the other monitors for the creation of *.p2* files. After running the application and adding and deleting a few files of the appropriate types, I get the results shown in Figure 6-18.

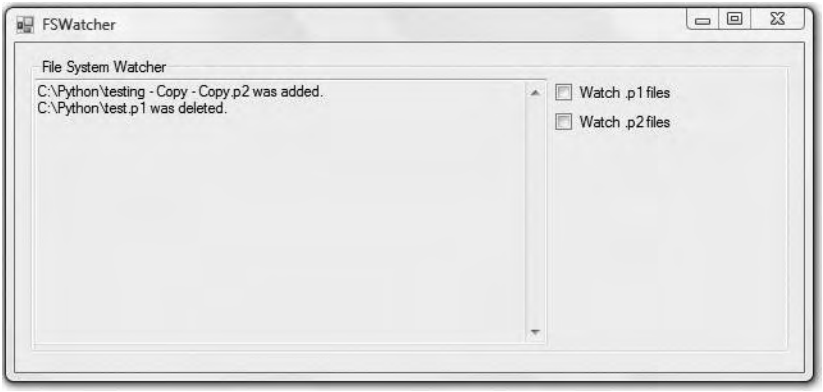


Figure 6-18. *The FSWatcher is using the IronPython plug-in code and monitoring the file system.*

So far so good. Now all we need to do is to check the state of the check boxes and make the correct processing decision based on their value (Listing 6-29).

Listing 6-29. *The Main Form with FileSystemWatcher Added*

```
private void timer1_Tick(object sender, EventArgs e)
{
    txtUpdates.Text = updateText;
    switch (p1WatcherEnabled.Checked)
    {
        case false:
            fp1.Deleted -= OnP1Deleted;
            break;
        default:
            fp1.Deleted += OnP1Deleted;
            break;
     }
    switch (p2WatcherEnabled.Checked)
    {
        case false:
            fp2.Created -= OnP2Created;
            break;
```

```
            default:
                fp2.Created += OnP2Created;
                break;
        }
    }
}
```

Now, if one or both of the check boxes are unchecked, the *FSWatcher* will not update the *txtUpdates* box with relevant file system data. The events are removed from the *FileSystemWatcher* and added back in when the boxes are checked (Figure 6-19).

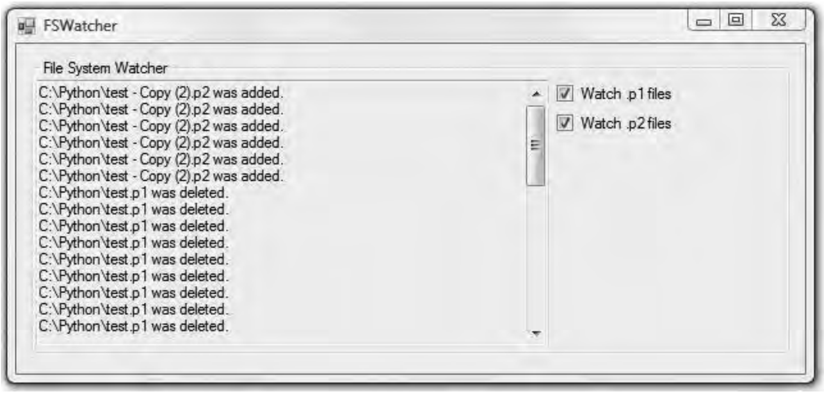


Figure 6-19. *The FSWatcher with adjusted settings*

Project Postmortem

Looking back, how did we do based on our design requirements?

1. The application should consist of a single Form. **DONE**.
2. The application should monitor the file system for various changes. **DONE**.
3. The application should handle all actions on file system changes via IronPython plug-ins. **DONE**.

Given that this application is technically simple, we did a pretty good job of putting our IronPython plug-in system through its paces. Hopefully you see there's a lot of potential for growth here. I purposefully kept the plug-ins simple, just returning some basic information. What we have accomplished is to create a functional plug-in system that can adapt to a variety of situations. We can pass and retrieve data from it, we've created a common format for plug-ins that permits some very consistent implementations, and your IronPython scripts can grow from there.

Summary

With the plug-in architecture in place, you've seen firsthand how quickly you can implement IronPython code in another .NET language. In fact, you can use the *IPEngine* library in any .NET language. Just add it as a reference, instantiate, and use it given the syntax of the language of your choice. That means you can drop your own IronPython code into any .NET application you write that makes use of *IPEngine*. We've looked at the constructions IronPython provides for integration (such as the *ScriptEngine*) and built a few applications that use IronPython plug-ins, including console and Forms applications. Although we took a brief detour into a land heavy with C# code, we're moving back to pure IronPython and we'll look at accessing various data sources.

CHAPTER 7

Data Manipulation

"Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build and test, it introduces security challenges and it causes end-user and administrator frustration."

— Ray Ozzie

You will not get far in development before you realize the need to store information that you are generating or collecting. Modern software applications typically store, or *persist*, their data somewhere for modification or retrieval later. There are a variety of ways to handle this task; databases, XML files, comma-separated value files, and flat text files are just some of the viable options, depending on the requirements of the application or organization. In this chapter we'll look at some of the various ways to store and retrieve data, beginning with one of the most common methods, the database. After covering a bit about each underlying data storage method, we'll look at how IronPython allows us to communicate with it to get our jobs done more easily.

SQL

When it comes to the task of data storage and retrieval, a **relational database management system** (RDBMS) such as Microsoft SQL Server is excellent at efficient data management. A database comprises one or more tables, with each table defining how to store information using columns of various data types. Data is stored as records, or rows in the tables, following the column specifications. Relating data in one table to that in another gives developers a powerful and versatile way to express data.

Note It's a common misconception that the word *relational* refers to the relationship between two or more tables. Actually, each individual table in the database is called a *relation*. The reason for this is that every row of data in the table conforms to the same constraints and data types; you can't have one row of *n* columns and another row with a different number of columns or data types. Knowing this won't make or break your database success, but it might serve to clarify certain aspects.

Consider that your employer has asked you to build an application to manage the employees in the company. Your boss wants an easily maintainable list of employees in a given department, along with some basic information, such as each employee's name, date of birth, and the date of hiring. A relational database design for this type of system might look something like Figure 7-1.

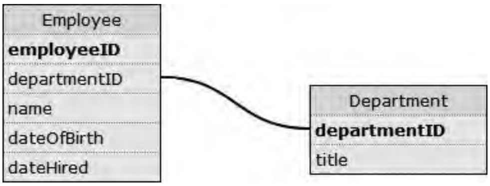


Figure 7-1. *A sample database design for the employees*

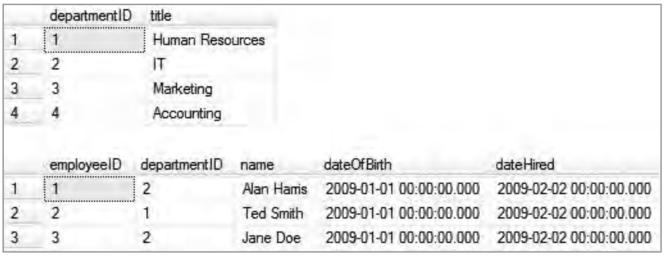
Figure 7-1 presents two tables: one called *Employee* and the other called *Department*. The Employee table has five columns, the Department table has two. Why are two of the columns in boldface? Those columns indicate keys. In the case of the Employee table, the *employeeID* column is what's called the *primary key*, which is a value used to identify uniquely each row in the table. The *departmentID* column in the Department table is a *foreign key*. The Department table defines the unique IDs for each department, and a relationship is established between the two tables. Basically, a foreign key is a stored reference to the primary key in another table. It is termed a foreign key because it references another table.

Tip Why didn't we just store the name of the department in the employee record? If we had, not only would it duplicate data unnecessarily, but it would make maintenance a nightmare. Where one user might enter "IT," another might enter "Information Technology," and, worse, yet another might enter "Developers." What happens when your boss asks for a list of everyone in the IT department?

By separating data in this manner, we have *normalized* it. Normalized data is designed to strengthen the integrity of the data in your system. And normalized data can take multiple forms. The table design for Employee and Department has been normalized to a point, but it could be normalized further. This is not always desirable, however. Most situations don't require normalization past what is called *3rd Normal Form*, and there are situations in which you'd actually like your data to be *denormalized* for performance reasons.

Where normalized data is very granular and generally requires several steps to assemble as a finished entity in an application, denormalized data could go so far as to have all the data you need stored in one row, making retrieval incredibly fast. Like many aspects of development, this is one of those trade-offs that often comes down to testing to find out what works best for the current problem.

Assuming that we have a few data points already filled in for both tables, Figure 7-2 demonstrates the contents and relationships in our sample company.



	departmentID	title
1	1	Human Resources
2	2	IT
3	3	Marketing
4	4	Accounting

	employeeID	departmentID	name	dateOfBirth	dateHired
1	1	2	Alan Harris	2009-01-01 00:00:00.000	2009-02-02 00:00:00.000
2	2	1	Ted Smith	2009-01-01 00:00:00.000	2009-02-02 00:00:00.000
3	3	2	Jane Doe	2009-01-01 00:00:00.000	2009-02-02 00:00:00.000

Figure 7-2. *Some sample data for our small company*

Note that the *departmentID* column has an integer value in it representing the department in which an employee works. Now, while it's all well and good that we can open these tables, it's not terribly efficient, because we have to look through the records manually and figure out who works where. Luckily, modern database systems provide a custom language for communicating with the system and working with data: Structured Query Language, or SQL (pronounced "sequel").

Although complete coverage of SQL is beyond the scope of this book, we're in luck because the language itself is built on very basic concepts, including some we have already covered. After setting up a basic design, we'll break down these four core operations and how to execute them via the IronPython console.

■ **Note** The .NET framework exposes a variety of providers and boilerplate code to make database connectivity simpler and to relieve you, as a developer, of the burden of creating data access code for MS SQL, Oracle, and other databases. Most of this functionality lives under *System.Data* and related namespaces that are specific to particular databases. For our purposes, we'll focus on *System.Data* and *System.Data.SqlClient*.

A Sample Database

First, we need to set up a sample database with some dummy data. We'll create the Employee and Department tables from earlier and fill them in with the data we've seen so far. Open up the Microsoft SQL Server Management Studio you installed in Chapter 1 and log in to the default instance (Figure 7-3).

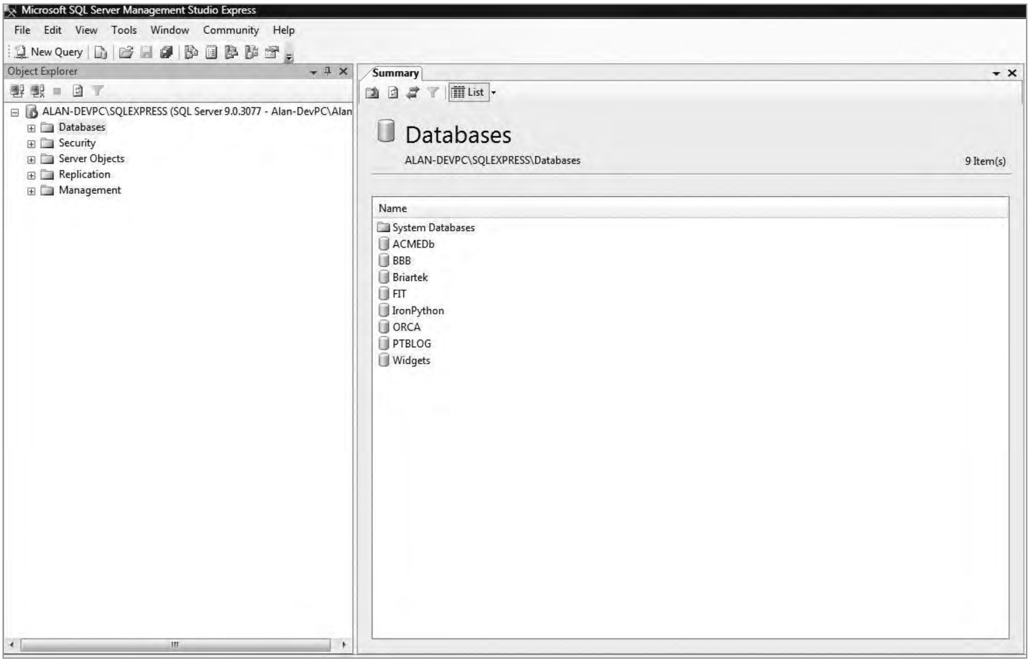


Figure 7-3. *Logging in to SQL Server*

On the left side you'll see the server to which you're currently connected; on my machine it's the SQLEXPRESS instance. Right-click on the *Databases* folder and click New Database. Name this new database *IronPython,* and leave the remaining options set to their defaults. Click OK to add the database (Figure 7-4).

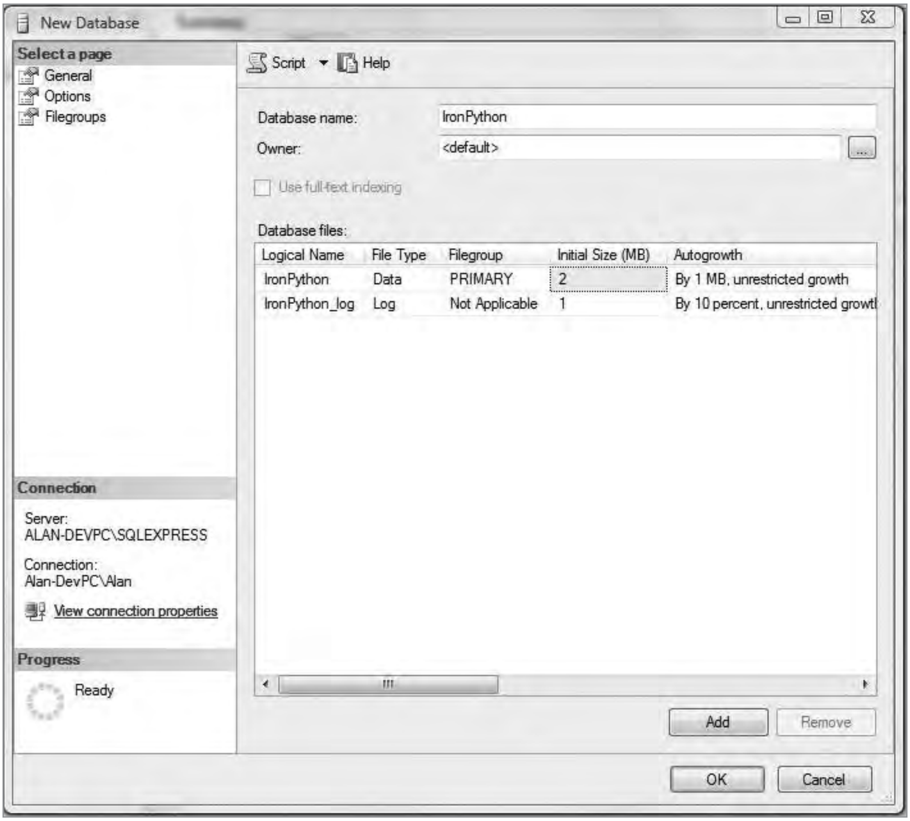


Figure 7-4. *Creating the IronPython database*

Next, we'll create the Department table. Right-click on the IronPython database and click New Query. Enter the query shown in Listing 7-1 in the Query Editor window on the right; press F5 to run it.

Listing 7-1. *SQL Code to Create the Department Table*

```
USE [IronPython]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
```

```
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[Department](
      [departmentID] [int] IDENTITY(1,1) NOT NULL,
      [title] [varchar](100) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
 CONSTRAINT [PK_Department] PRIMARY KEY CLUSTERED
([departmentID] ASC)
WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
```

■**Note** The SQL statements for creating and modifying a table's structure, keys, indexes, and relationships can be complex, so don't spend too much time trying to understand these statements at the moment. They fall into a category of SQL statements called *data definition language*, or *DDL*, whereas most of our time will be spent with *data modification language,* called *DML*. The DML statements can get very, very complex as well, but we won't dive into the deep end.

Now we'll create the Employee table; then we can start populating everything with data. Replace the previous script with the one in Listing 7-2 and execute it by pressing the F5 key.

Listing 7-2. *SQL Code to Create the Employee Table*

```
USE [IronPython]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[Employee](
      [employeeID] [int] IDENTITY(1,1) NOT NULL,
      [departmentID] [int] NOT NULL,
      [name] [varchar](150) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
      [dateOfBirth] [datetime] NOT NULL,
      [dateHired] [datetime] NOT NULL,
```

```
 CONSTRAINT [PK_Employee] PRIMARY KEY CLUSTERED
([employeeID] ASC)
WITH (PAD_INDEX  = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
GO
SET ANSI_PADDING OFF
GO
ALTER TABLE [dbo].[Employee]  WITH CHECK ADD  CONSTRAINT [FK_Employee_Employee]➥
FOREIGN KEY([departmentID])
REFERENCES [dbo].[Department] ([departmentID])
GO
ALTER TABLE [dbo].[Employee] CHECK CONSTRAINT [FK_Employee_Employee]
```

Finally, replace that SQL script with the one in Listing 7-3 and execute it to populate the tables with a little bit of data.

Listing 7-3. *SQL Code to Fill the Tables with Default Values*

```
INSERT INTO Department (title) VALUES ('Human Resources')
INSERT INTO Department (title) VALUES ('IT')
INSERT INTO Department (title) VALUES ('Marketing')
INSERT INTO Department (title) VALUES ('Accounting')

INSERT INTO Employee VALUES (2, 'Alan Harris', '1/1/2009', '2/2/2009')
INSERT INTO Employee VALUES (1, 'Ted Smith', '1/1/2009', '2/2/2009')
INSERT INTO Employee VALUES (2, 'Jane Doe', '1/1/2009', '2/2/2009')
```

Create

Adding new records (or rows) to a SQL table from your program can be accomplished with the INSERT INTO command. Let's try adding a record to the Department table from IronPython. Open the IronPython interpreter and enter the code in Listing 7-4—replacing the `ALAN-DEVPC\SQLEXPRESS` name in the `sqlConnection` with the name of your SQL instance.

Listing 7-4. *Adding a Department to the Department Table*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
```

```
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> comm = SqlCommand("INSERT INTO Department VALUES ('IPY Department')", conn)
>>> conn.Open()
>>> comm.ExecuteNonQuery()
1
>>> conn.Close()
```

■ **Note** Immediately after typing the `ExecuteNonQuery()` command, you should see the number 1 returned from the IronPython interpreter. This is SQL Server informing you of how many rows were modified by the last statement. In this case, you inserted one record, so the return value says that you modified one row in that statement.

The INSERT INTO command needs to know which table to insert data to, as well as what data to insert. Note that in SQL commands, strings are denoted by single quotes, so if you want to insert a single quote into a field (for example, a department called *Joe's Department*), you would provide a value of *Joe''s Department*. Also note that the fields in the INSERT statement are in the order in which they appear in the table; the first column is left off in our case because SQL Server will fill it in automatically, since it's set to be an auto-incremented primary key.

Let's add an entry to the Employee table that demonstrates this (Listing 7-5). Remember to change the `SqlConnection` to specify your database.

Listing 7-5. *Adding a Department to the Employee Table*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> comm = SqlCommand("INSERT INTO Employee VALUES (1, 'Sally''s Employee',➥
'1/1/2009', '2/2/2009')", conn)
>>> conn.Open()
>>> comm.ExecuteNonQuery()
1
>>> conn.Close()
```

Here we've added an employee to the Employee table who is a member of the Human Resources Department (department ID 1) and who is named *Sally's Employee*. She was born on January 1, 2009, and hired on February 2, 2009 (which *has* to violate a law somewhere).

Retrieve

If we want to retrieve data from a SQL table, we can use the SELECT command. There are multiple ways to use the SELECT command. You can specify the columns you want to retrieve, as in Listing 7-6, or you can select all the columns in the schema, as in Listing 7-7.

Listing 7-6. *Selecting Specific Columns from the Employee Table*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> comm = SqlCommand("SELECT name FROM Employee", conn)
>>> conn.Open()
>>> reader = comm.ExecuteReader()
>>> if (reader.Read()):
        print (reader.GetString(0))
    (press Enter here)
Alan Harris
>>> reader.Close()
>>> conn.Close()
```

■**Tip** There are a variety of typed methods for the SqlDataReader, such as *GetString*, *GetChar*, and *GetBoolean*. These are the fastest methods of retrieving data from the database, but they require your code to have an intimate knowledge of the underlying data types of the table. Depending on your situation, this may or may not matter. For the sake of performance, this is the manner of data retrieval we'll use.

Listing 7-7. *Selecting All Records from the Employee Table*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> comm = SqlCommand("SELECT * FROM Employee", conn)
>>> conn.Open()
>>> reader = comm.ExecuteReader()
>>> if (reader.Read()):
        print (reader.GetString(2))
    (press Enter here)
Alan Harris
>>> reader.Close()
>>> conn.Close()
```

■ **Tip** Although it can be tempting, don't rely on SELECT * in any real-world code you plan to use. It is wasteful because it can potentially return unnecessary columns in the result set, it requires the retrieval of the schema of the database to map columns in place of the wildcard, and it makes maintenance a potential pain if something breaks in the schema. Generally speaking you are safest if you are explicit about each of the column names in your query. Also note that we had to move to a different column in the result set the second time because all of the columns had been returned to us. Tacky!

The SqlDataReader we used to read the data from the table is a forward-only stream of data; it's not possible to move backward in the stream. The SqlDataReader is a very fast method of data retrieval, though, and it is used very commonly.

Update

Modifying existing data in a SQL table is done via the UPDATE command. To use the UPDATE command, one or more rows in the table must already exist to be updated, or the command will fail. Let's update the record for "Sally's Employee," as shown in Listing 7-8.

Listing 7-8. *Updating the Record for "Sally's Employee" in the Employee Table*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> comm = SqlCommand("UPDATE Employee SET name='Sally Employee'➥
WHERE name='Sally''s Employee'", conn)
>>> conn.Open()
>>> comm.ExecuteNonQuery()
1
>>> conn.Close()
```

When executing statements that modify or delete data, it's particularly important to make sure you've applied the WHERE clause properly, otherwise you'll modify every row in the table! I can think of a few cases where this is the intended behavior, but not many. You can also use the WHERE clause to limit the results you get back in SELECT statements.

■ **Note** If there is no data to update that matches the WHERE clause filter (if any), you'll just get 0 rows returned. Technically, the command has failed, but it's not a catastrophic failure, just a logical one.

Delete

Deleting data from a SQL table is reserved for the aptly named DELETE command. Having had both a short and tumultuous career in our small company, it's time to delete the name-changing "Sally Employee" (Listing 7-9).

Listing 7-9. *Updating the Record for "Sally's Employee" in the Employee Table*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
```

```
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> comm = SqlCommand("DELETE FROM Employee WHERE name='Sally Employee'", conn)
>>> conn.Open()
>>> comm.ExecuteNonQuery()
1
>>> conn.Close()
```

■**Note** If you want to delete everything in a table and reset any incremental keys back to 0 in one step, you can use the broader command TRUNCATE TABLE. However, this does not allow you to delete individual rows; it will eradicate *all* the data in the table. If you wanted to clear out the Employee table, you could use *TRUNCATE TABLE Employee*. Don't execute that command unless you want to obliterate all the data in your table!

Preventing SQL Injection Attacks

So far we've operated in a very safe, sterile data environment with SQL. There has been no user-supplied input to sanitize. This is generally not representative of the world we live in as developers. I want you to pause for a moment and digest this next sentence carefully and deliberately. It's important. Consider it the most important quote of this chapter.

> *"All input should be considered dangerous and never trusted. No exceptions."*

This sounds like a terribly pessimistic view of the world, but it's better to treat all input as dangerous than to get caught with your security lacking. There are developers who believe that SQL injection attempts are not common or are not as easy to perform as some would have you think; this is absolutely untrue. Burying your head in the sand does not relieve you of the threat.

Listing 7-10 is an example of code that's very susceptible to injection; I've made the dangerous input bold for clarity.

■**Caution** Listing 7-10 is for example purposes ONLY! Don't run this code directly unless you're comfortable with losing the data in the Employee table. Everything we currently have is testing data only and can easily be recreated, but I want you to know what you're in for before you execute dangerous code.

Listing 7-10. *A 5cc Injection of SQL Insecurity*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> empName = raw_input('Enter the employee name to delete: ')
Enter the employee name to delete: foo' OR name IS NOT NULL;
>>> comm = SqlCommand("DELETE FROM Employee WHERE name='" + empName + "'", conn)
>>> conn.Open()
>>> comm.ExecuteNonQuery()
3
>>> conn.Close()
```

We didn't perform any steps to sanitize the user input. By not doing so, we allowed the execution of arbitrary SQL commands and left our database exposed.

The final statement we issued to SQL Server looks like Listing 7-11.

Listing 7-11. *What We Really Told the Database to Do*

```
DELETE FROM Employee WHERE name='foo' OR NAME IS NOT NULL;
```

You can see that we told SQL Server to delete everything in the Employee table where the name was set to *foo* or the name was not null (thus basically every employee with a name!). SQL injection attacks can be particularly insidious, and they tend to hide in the nooks and crannies of legacy code.

What's a .NET developer to do? You can opt to try to clean all your SQL inputs yourself by hand (which many developers have done over the years, to varying degrees of success), or you can use the tried-and-tested parameterized SQL statements that .NET provides.

Parameterized Queries

Using parameterized queries means letting go of assembling SQL statements involving string concatenation. If you're new to development, then you've nothing to worry about because you've got no bad habits to break. If you're used to building SQL statements this way, watch how easily you gain some security over your statements, with minimal work (Listing 7-12).

Listing 7-12. *With Parameterized Queries, We Add a Nice Layer of Security to Our Code*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> empName = raw_input('Enter the employee name to delete: ')
Enter the employee name to delete: foo' OR name IS NOT NULL;
>>> comm = SqlCommand("DELETE FROM Employee WHERE name=@name", conn)
>>> conn.Open()
>>> comm.Parameters.AddWithValue("@name", empName)
<System.Data.SqlClient.SqlParameter object at 0x000000000000002B [@name]>
>>> comm.ExecuteNonQuery()
0
>>> conn.Close()
```

Instead of using concatenation, we added to the statement a parameter called `@name`. Then we supplied a value for the parameter. Behind the scenes, .NET did some work to escape various characters and patterns in the statement so that everything would be treated as string literals and not as interpreted commands. As a result, instead of deleting all the rows in the table, none of them were deleted. It's a small price to pay for security, wouldn't you agree?

■ **Caution** Don't fall into the trap of assuming that because you've followed basic guidelines for security that you're immune to all attacks. You're definitely better off for using the resources .NET provides, but security should be one of your primary concerns at all times during application development. Any minor flaw can be exploited with disastrous results.

Stored Procedures

In terms of performance and security, it's hard to argue against stored procedures. The SQL commands we've run so far have been inline SQL; that is, they have been complete sets of SQL instruction contained entirely within the IronPython source code itself. Some developers swear by this, others swear against it. It's a holy war in and of itself. Let's weigh the pros and cons before looking at how it's actually done.

When SQL Server executes queries, a lot of work is going on behind the scenes to optimize the result set and to cache the execution plan. By placing your code in a stored procedure, you help SQL Server perform those optimization steps and reduce the load on the database. Maintenance of your code is generally easier because SQL code modifications can be made server-side without requiring recompilation or deployment of updated code modules; any client code that uses the stored procedures will automatically use the updated SQL code on the next connection. Finally, you can perform more steps on the server itself, reducing the traffic to and from your program and the server and allowing the database to do its job.

On the other side of the coin, keeping smaller bits of SQL code in your source code can help to keep the database from getting cluttered, which becomes important if you're working on an enterprise application where the number of stored procedures can be in the hundreds. From a design standpoint, it keeps potential business domain code in the business layer and not out at the database itself. Frankly, it can be easier to keep your SQL in your source code because it means you don't have to switch back and forth between SQL Server and your code IDE. When a deadline looms, sometimes every second counts and you don't want to waste that time moving between applications.

My personal preference is for stored procedures in most cases and for inline SQL in limited other cases. In general, if you've got complex code or code that is going to be executed very frequently, place it in a stored procedure and let SQL Server manage it as it sees fit. If you've got a small bit of infrequently hit code that isn't likely to change in the near future, it won't kill you to have it in your source code. I will say that making these distinctions is mainly a judgment call based on experience. If you're not sure or the answer isn't crystal clear, place your SQL in a stored procedure, because this is the safer choice. Be aware, however, that when rolling code out to production, you have to ensure you get those stored procedures to your production database first! With inline SQL, this is not a concern.

■ **Tip** For a long time, the recommended naming convention for stored procedures was that they begin with *sp*. So if you were to make a stored procedure that logged in a user, you would likely name it *spLoginUser*. This is a bad practice to get into; SQL Server will check the system procedures first based on that prefix and you'll incur a negligible performance penalty for it. It's nothing major, but if you're blessed with millions of happy, active users connecting to your data storage, every little bit counts and that's a silly one to waste any resources on.

I would recommend prefixing your stored procedures based on the area of your program to which they're related or on the function they provide. For example, for a login stored procedure, I might call it *securityLoginUser* or *authenticationLoginUser*. The trick really is to be consistent. Don't have *securityLoginUser* and *authenticationLogoutUser* if you can help it. It's just a recipe for confusion at that point.

Let's create a simple stored procedure to retrieve an employee's name; then we'll see how to use it from IronPython. Create a new query in SQL Server and enter the code shown in Listing 7-13.

Listing 7-13. *A Simple Stored Procedure to Retrieve an Employee's Name*

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE GetEmployeeNameByID
      @employeeID INT
AS
BEGIN
      SET NOCOUNT ON;
      SELECT [name] FROM Employee
      WHERE employeeID = @employeeID
END
GO
```

We can try it out in SQL Server before calling it from our IronPython code with the line of code in Listing 7-14.

Listing 7-14. *Calling the Stored Procedure with a Sample ID*

```
EXEC GetEmployeeNameByID 1
```

You should get a single result; for me it was my name, *Alan Harris*.

Now let's call this stored procedure from IronPython (Listing 7-15).

Listing 7-15. *Selecting Records from the Employee Table with a Stored Procedure*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> comm = SqlCommand("GetEmployeeNameByID", conn)
>>> comm.CommandType = CommandType.StoredProcedure
```

```
>>> conn.Open()
>>> comm.Parameters.AddWithValue("@employeeID", 1)
<System.Data.SqlClient.SqlParameter object at 0x000000000000002C [@employeeID]>
>>> reader = comm.ExecuteReader()
>>> if (reader.Read()):
        print (reader.GetString(0))
    (press Enter here)
Alan Harris
>>> reader.Close()
>>> conn.Close()
```

Tip Earlier in the book I mentioned that certain resources were expensive and needed to be freed properly. Database connections and resources can be very expensive and are prone to leaks due to easily missed coding flaws. Always make sure to close your *SqlDataReaders* and *SqlConnections,* or you'll see things grind to a halt very quickly. It's easy to overlook and can cost you plenty in hard-to-diagnose performance problems and dropped connections.

Connection Pooling

The act of creating and destroying connections to the database is very costly, from a performance perspective. It takes a heavy toll on response times and, if performed too frequently, can completely disable a database or application as SQL Server struggles to keep up with resource demands. To counteract this problem, connection pooling was created.

When you create a connection to SQL Server with connection pooling enabled, then if there are any existing connections in the pool, one will be retrieved and reused. This is a very quick procedure. SQL Server maintains a cache of these connections, so the overhead of creating a connection object is not incurred.

Tip Connection pooling is on by default if you do not specify otherwise, but it never hurts to be explicit. I find that fewer errors occur if you take the time to specify what you want outright. A wasted day of work is one spent troubleshooting a poor SQL connection that is flaking out because of connection pooling issues.

The trick to connection pooling is that the pooling works only if the connection string is the same every time you communicate with the database. If even a single character is

different, then the connection string is not identical and a new connection will be created. For that reason, I highly recommend that, in your data access code, you create a variable to hold the connection string (Listing 7-16) and that you reuse that variable when you need it. That will help prevent typographical errors and ensure that your database connections run smoothly.

Listing 7-16. *A Sample Connection String Variable*

```
>>> connString = "Data Source=ALAN-DEVPC\SQLEXPRESS;➥

Integrated Security=True;Initial Catalog=IronPython;User Instance=false;➥
Max Pool Size=100;Min Pool Size=5;Pooling=true"

>>> conn = SqlConnection(connString)
```

■ **Tip** A maximum pool size of 100 and a minimum pool size of 5 are also default values in .NET and perfectly reasonable values to use in production.

XML

As an alternative to databases for the storage and retrieval of data, developers can opt to use XML, short for Extensible Markup Language, as their weapon of choice. The interesting thing about XML is that its technical purpose is to facilitate the creation of *other* markup languages. Before we dive into working with XML, let's take a look at what some XML markup looks like (Listing 7-17).

Listing 7-17. *A Sample XML Document*

```
<?xml version="1.0" ?>
<documents>
     <document type="book">
          <author>Alan Harris</author>
          <title>Pro IronPython</title>
          <pubYear>2009</pubYear>
     </document>
```

```
    <document type="paper">
        <author>King Kong</author>
        <title>How to Climb a Building</title>
        <pubYear>1933</pubYear>
    </document>
</documents>
```

It's actually rather simple. However, there are some rules to properly formatted XML that are important to follow.

- Tags must be well formed. That means opening and closing brackets, **< and >,** as well as opening and closing tags. If you open an element **<dog>**, there must be a matching **</dog>** to close that element.
- Tags are case sensitive; **<dog>** cannot be properly closed by a tag of **</Dog>**.
- Tags must not overlap, meaning that the tree hierarchy must be preserved. **<dog><cat></cat></dog>** is valid, but **<dog><cat></dog></cat>** is not. Tags opened within other tags must be closed within that same tag.

As with stored procedures, there is some debate over the pros and cons of XML. Proponents of XML say that it's a terrific, well-structured way to convey **metadata**. It lends itself to standards and can easily be validated because of its fairly strict rule set. On the other side of the fence, there are those who view XML as a failed experiment. They consider it a tremendously verbose way to describe data and feel it is prone to errors during transmission. Regardless of which side you're on, it remains an oft-used method of communication and storage. In fact, in the next chapter, in which we work with web development, we'll see that not only does the .NET framework whole-heartedly embrace XML for its configuration data, but the basic construction of a web page can be expressed in a way that is very similar to XML.

■ **Note** *Metadata* is, quite literally, data that describes other data. In Listing 7-17, the XML that describes documents is a good example of metadata; the XML describes data points about other objects.

To see a real-world example of working with XML, let's use the RSS feed from *USA Today*. At the time of this writing, the feed was available at `http://rssfeeds.usatoday.com/usatoday-NewsTopStories`. The first step is to write a little IronPython code to retrieve this feed (Listing 7-18).

Listing 7-18. *Retrieving an RSS Feed*

```
>>> import clr
>>> clr.AddReference("System")
>>> clr.AddReference("System.Net")
>>> from System import *
>>> from System.Net import *
>>> from System.IO import *
>>> request =➥
WebRequest.Create("http://rssfeeds.usatoday.com/usatoday-NewsTopStories")
>>> response = request.GetResponse()
>>> reader = StreamReader(response.GetResponseStream())
>>> result = reader.ReadToEnd()
```

The variable called `result` now contains the entire XML markup of the RSS feed. You can verify this for yourself. Note that I did not display the entire contents here, for the sake of brevity; you will see significantly more output from printing the `result` variable (Listing 7-19).

Listing 7-19. *Retrieving an RSS Feed (cont.)*

```
>>> print result.ToString()
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:feedburner="http://rssnamespace.org/feedburner/ext/1.0"➥
xmlns:cf="http://www.microsoft.com/schemas/rss/core/2005" version="2.0">
  <channel>
    <cf:treatAs>list</cf:treatAs>
    <title>USATODAY.com News - Top Stories</title>
    <link>http://www.usatoday.com/news/default.htm</link>
    <description>USATODAY.com News - Top Stories (USA TODAY)</description>
    <language>en-us</language>
    <copyright>Copyright 2009, USATODAY.com, USA TODAY</copyright>
    <lastBuildDate>Mon, 09 Mar 2009 22:07:00 GMT</lastBuildDate>
    <image>
      <title>USATODAY.com News - Top Stories</title>
```

Now we have the XML, but it is in string format, which isn't terribly useful for navigating; it's a rather flat, meaningless structure. We could use some string parsing methods to try and extract the data we want, but that would be a clunky, unnecessary step. We should treat it as proper XML and use the navigation methods .NET provides (Listing 7-20).

Listing 7-20. *Retrieving an RSS Feed (cont.)*

```
>>> clr.AddReference("System.Xml")
>>> from System.Xml import *
>>> xmlReader = ➥
XmlTextReader("http://rssfeeds.usatoday.com/usatoday-NewsTopStories")
>>> while (xmlReader.Read()):
        print xmlReader.NodeType.ToString() + " " + xmlReader.Name
    (press Enter here)
```

The *XmlTextReader* is the XML equivalent of the *SqlDataReader*. It is a forward-only reader that is very fast but that does not allow modification of the underlying data. It is only for efficient retrieval. What if we want to write an XML document to disk (Listing 7-21)?

Listing 7-21. *Writing an XML File to Disk*

```
>>> import clr
>>> clr.AddReference("System")
>>> clr.AddReference("System.Xml")
>>> from System import *
>>> from System.Xml import *
>>> from System.IO import *
>>> xmlWriter = XmlTextWriter("C:\Python\IPXml.xml", Text.Encoding.UTF8)
>>> xmlWriter.WriteProcessingInstruction("xml", "version='1.0' encoding='UTF-8'")
>>> xmlWriter.WriteStartElement("dog")
>>> xmlWriter.WriteStartElement("cat")
>>> xmlWriter.WriteEndElement()
>>> xmlWriter.WriteEndElement()
>>> xmlWriter.Close()
```

Opening this file in Notepad shows the result in Figure 7-5, indicating that we were successful in storing data to the drive.

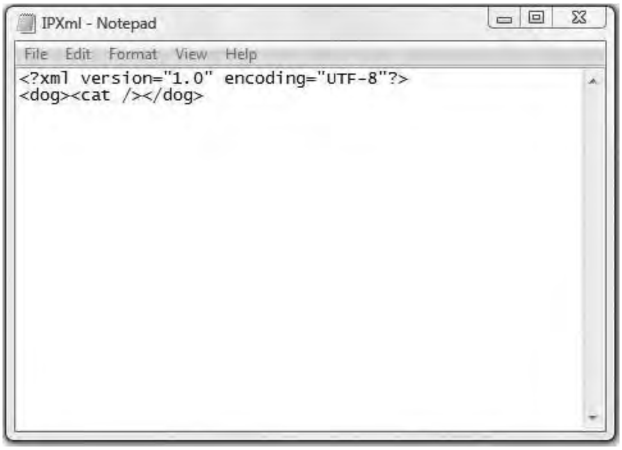


Figure 7-5. *The XML file was created successfully.*

Comma-Separated Values

Comma-separated values (CSV) are a very simple way of storing data. In a way they are very similar to databases; that is, they can be thought of in terms of rows and columns. Each row of text is expressed as one line; the columns are separated by a comma.

■ **Note** Technically speaking, it doesn't *have* to be a comma that separates values. It could be any character you want to parse out. But obviously if someone is expecting you to work with traditional CSV data, that person is going to expect the C to stand for *comma*.

I created a simple text file that contains some CSV data and saved it to my desktop (Figure 7-6). You'll see that consuming this information in IronPython is extremely trivial (Listing 7-22).

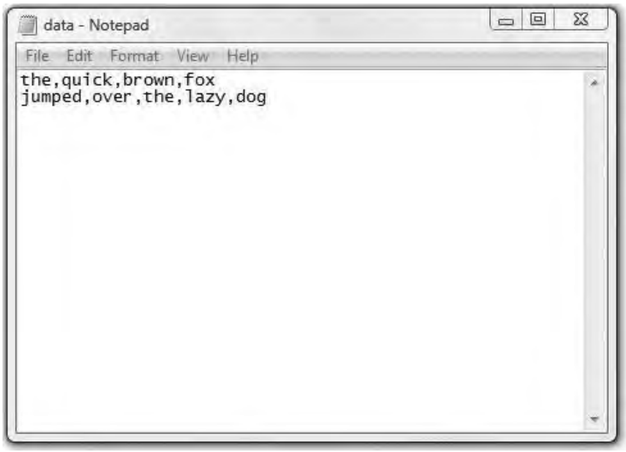


Figure 7-6. *A simple CSV file*

Listing 7-22. *Reading and Displaying the Contents of a CSV File*

```
>>> import clr
>>> clr.AddReference("System")
>>> from System.IO import *
>>> lines = File.ReadAllLines("C:\Python\data.csv")
>>> for line in lines:
       fields = line.Split(',')
       for field in fields:
        print field
    (press Enter here)
the
quick
brown
fox
jumped
over
the
lazy
dog
```

Creating an Effective Data Layer

As I've mentioned all along, it's important when building applications that you separate concerns and keep code divided into easily maintainable layers. Before now this has been limited exclusively to the presentation and business layers, but the third component to the three-tier architecture is the data layer. Building an effective, easily maintained data layer doesn't have to be a nightmare; in fact it can be downright simple, if you plan carefully.

The first concern is weighing the likelihood that the underlying data storage platform will change. .NET has a variety of providers for different data sources, but so far we've tied ourselves to a particular one (MS SQL Server.) That's okay if it's not likely that our data storage will change. I myself have no plans to shift from SQL Server on my local machine, so I'm not building a data layer to accommodate that. However, you may find that you need to, particularly if you plan to build public components where you don't know exactly what platform someone is going to be working with.

You also need to make the design decision of where to store your SQL code. Earlier we weighed the pros and cons about whether or not it was better to create stored procedures or to go with inline SQL. Again, I would recommend stored procedures where possible. But if you have infrequently hit small bits of code, the world won't come crashing down if you leave them in your code.

So what makes an effective data layer? In my mind, an effective data layer does its job when it facilitates easy, straightforward access to the underlying data storage without revealing an unnecessary amount of implementation details to any calling layers.

What this means is simply that the business logic shouldn't know that we're using SQL Server. That information is *none of its business*. Think of your layers like noisy neighbors; you should be instructing them to mind their own business and to perform their own individual tasks. The data layer should handle all the implementation details of speaking to the data storage application and just quietly pass the end result of that work to the business layer for appropriate processing. For comparison, we will examine some methods that use stored procedures and some that use inline SQL so that you can make an informed choice.

■ **Note** This is where the data layer holy war grumbling begins. Should the application-stored procedures do a minimalistic, straightforward data retrieval and leave *all* processing to the business layer, even if it means retrieving and returning unnecessary data? What exactly constitutes business logic, and does it have any place in stored procedures? These are not easy questions to answer. In my mind, the decision comes down to maintainability. If code is obviously difficult to maintain in a particular location, it's a likely candidate for being moved elsewhere. You may find that moving it to another layer is a bad idea or does not improve the maintainability of the application, but never be afraid to bend the rules a bit to find what works best for your particular application.

Let's try designing a data layer from scratch. We'll create a new Windows Forms project called *IPData*; make sure the *Create directory for solution* box is checked, as usual. Once the project is created, right-click on the project and add a folder called *Data*. Next, add a new class file to this *Data* folder. We'll call this file *dataManager.py*. The code for this class is given in Listing 7-23.

Listing 7-23. *The dataManager.py Class*

```
import clr
clr.AddReference("System")
clr.AddReference("System.Data")
from System import *
from System.Data import *
from System.Data.SqlClient import *

class dataManager:
    "The data layer manager for our IronPython test application"

    def GetConnection(self):
        "Gets a new SqlConnection object"
        conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=False;➥
Max Pool Size=100;Min Pool Size=5;Pooling=True")
        return conn

    def GetCommand(self, commandString, connection):
        "Gets a new SqlCommand object"
        comm = SqlCommand(commandString, connection)
        return comm
```

Note that this also fulfills the connection pooling requirements we had from earlier. By defining a *GetConnection* method that returns the same connection string each time, we should have no problems with connection pooling (and if the settings here should ever change in the future, we only have to change them in this one place, which will make maintenance simpler).

Using the dataManager

Now we can create an *employeeData* class in the *Data* folder that makes use of the *dataManager* class to handle the connectivity aspects of data access (Listing 7-24).

Listing 7-24. *The employeeData.py Class*

```
import clr
clr.AddReference("System")
clr.AddReference("System.Data")
from System import *
from System.Data import *
from System.Data.SqlClient import *
from dataManager import *

class employeeData:
    "The data class for the Employee table."

    def GetEmployeeNameByID(self, employeeID):
        result = String.Empty
        dm = dataManager()
        conn = dm.GetConnection()
        comm = dm.GetCommand("GetEmployeeNameByID", conn)
        comm.CommandType = CommandType.StoredProcedure
        conn.Open()
        comm.Parameters.AddWithValue("@employeeID", employeeID)
        reader = comm.ExecuteReader()
        if (reader.Read()):
            result = reader.GetString(0)
            reader.Close()
            conn.Close()
            return result
```

So let's look at what's happening in the *GetEmployeeNameByID* method. First, we created a variable called *result* that will hold the output of the method, if any. Next, we instantiated a *dataManager* class. Next we created a connection, followed by a command. Notice that we're passing a SQL statement (either a series of commands or the name of a stored procedure) and the connection itself to the *GetCommand* method. Next, we explicitly told the command object that what we provided should be found as a stored procedure. If you have a stored procedure but do not perform this step, the procedure will not be executed.

■ **Note** The reverse is not true: if you supply a SQL statement (such as `SELECT * FROM Employee`), you don't need to specify anything in particular for the *CommandType* parameter. Only if you're instructing SQL Server that the command text is the name of a stored procedure do you need to take that extra step.

Next we added the necessary parameters to the command object and provided the appropriate value. We retrieved the result from the table and returned it via the result string. Now we can call our *employeeData* class—can't we? Technically, yes. But developer to developer, no, we can't. We can't call it because the only place from which we could call it is the user interface directly, and we've established that as a no-no.

Business As Usual

Add a folder to the application called *Business*, and inside create an *employeeBusiness* class (Listing 7-25).

Listing 7-25. *The employeeBusiness.py Class*

```
from employeeData import *

class employeeBusiness:
    "The business class for the Employee table."

    def GetEmployeeNameByID(self, employeeID):
        emData = employeeData()
        return emData.GetEmployeeNameByID(employeeID)
```

Admittedly, the *GetEmployeeNameByID* method in the *business* class doesn't do much other than act as a pass-through. *That's not a bad thing*. I've heard people argue against this, but what I find is that having this type of setup gives you options. Sure, it's not functioning as anything other than a pass-through at this point. But if that requirement changes down the line, you already have hooks in your code on which to hang additional code. If you choose a simpler, more direct route, you'll have to restructure multiple locations to make even basic changes (maybe you need to validate the *employeeID* value here to ensure it's an integer and not a string).

Now we can modify the main form to accept some user input so that we can retrieve these fields dynamically. Add a text box and a button to the form. Name the text box *txtEmployeeID* and the button *btnSubmit*. Set the button's text to be *Get Name*. Double-click on the button and add the code in Listing 7-26.

Listing 7-26. *The Submit Button with Code Wired In*

```
@accepts(Self(), System.Object, System.EventArgs)
@returns(None)
def _btnSubmit_Click(self, sender, e):
    employee = employeeBusiness()
    self._txtEmployeeID.Text =➥

employee.GetEmployeeNameByID(self._txtEmployeeID.Text)
```

You'll also need to import the *employeeBusiness* class at the top of the form. Now if you run the application, you can enter an employee ID in the text box, and when you click the button, the text will be swapped with the employee name (Figure 7-7).

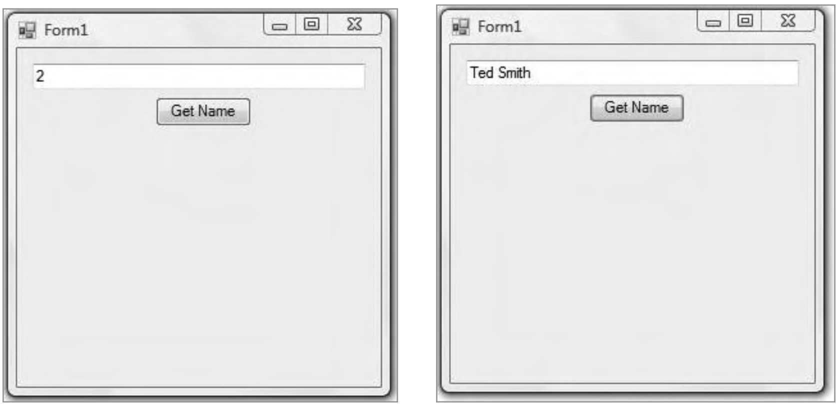


Figure 7-7. *The data layer is accurately returning our fields based on user input.*

All is not entirely well, however. Watch what happens when you click the Get Name button again without entering a number instead of the employee name (Figure 7-8).

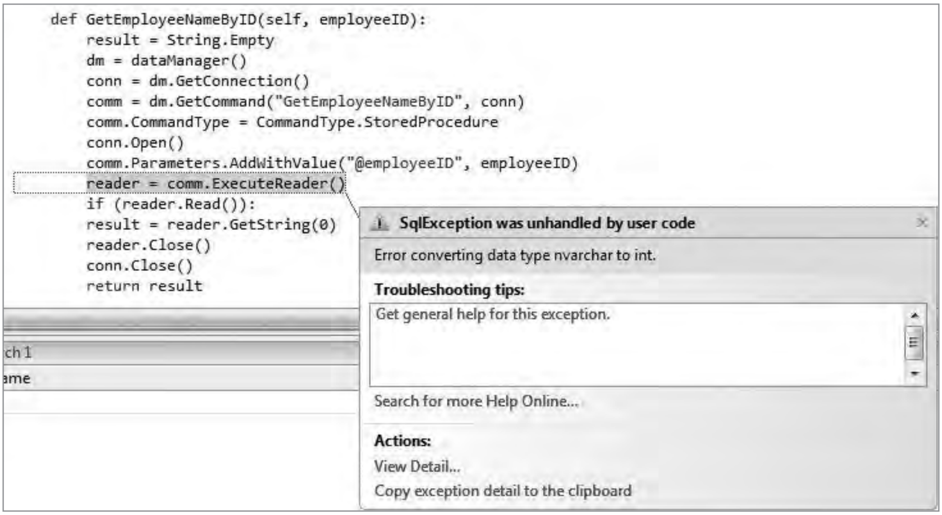


Figure 7-8. *Whoops! This is how connections leak. Notice that we're not hitting the Close methods.*

Exceptional Handling!

We have several options for places to step in and do a little data validation. But, regardless of what we do, one step *has* to be ensuring that connections get closed and resources released in the event of an exception. So let's modify that *employeeData* class with some exception-handling blocks (Listing 7-27, Figure 7-9).

Listing 7-27. *Modifying the employeeData Class with Exception Handling*

```
import clr
clr.AddReference("System")
clr.AddReference("System.Data")
from System import *
from System.Data import *
from System.Data.SqlClient import *
from dataManager import *
```

```
class employeeData:
    "The data class for the Employee table."

    def GetEmployeeNameByID(self, employeeID):
        result = String.Empty
        dm = dataManager()
        conn = dm.GetConnection()
        comm = dm.GetCommand("GetEmployeeNameByID", conn)
        comm.CommandType = CommandType.StoredProcedure
        try:
            conn.Open()
            comm.Parameters.AddWithValue("@employeeID", employeeID)
            reader = comm.ExecuteReader()
            if (reader.Read()):
              result = reader.GetString(0)
              reader.Close()
              conn.Close()
              return result
        except:
              conn.Close()
        if (conn.State == ConnectionState.Open):
              conn.Close()
              return result
```

■ **Tip** What's the deal with calling `conn.Close()` so many times? Well, the simple answer is that there's no penalty or risk in calling it multiple times. The *Close* method won't throw an exception if it is called after the connection has been closed; it just quietly passes along. Technically speaking it should never be called more than twice in this code anyway, because the final Close is wrapped in a check against the current state of the connection.

If you run the application again, feel free to try submitting anything you like. The database will return a valid result only for employee IDs that exist in the table. Anything that doesn't match that criterion will result in a blank value in the text box.

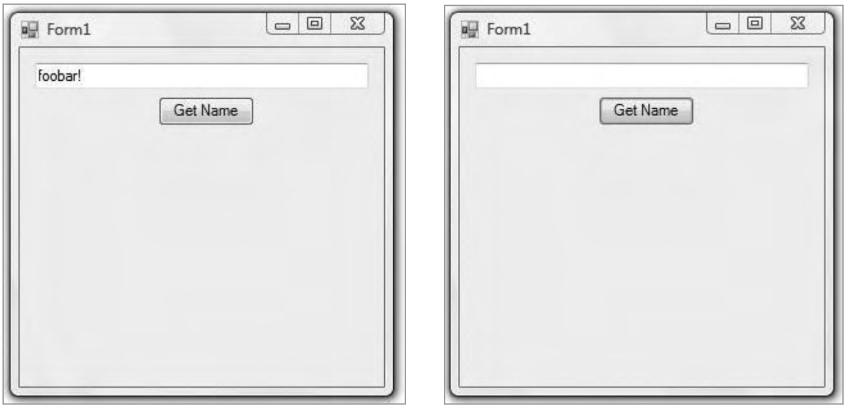


Figure 7-9. *With exception handling in place, the program no longer fails catastrophically.*

■ **Tip** Exceptions, by definition, are meant to handle exceptional, fringe cases. You shouldn't leave operation of the program up to the exceptions you've created. They're fairly costly operations, despite being very fast in .NET. A simple check in an earlier layer would prevent the overhead of setting up the SQL connection and parameters, handling the exception, cleaning up, and so on.

Inserting a New Employee

Now let's try adding a new employee to the table. We're going to hard-code the department for simplicity's sake, but at this point adding one additional parameter should be fairly straightforward. In the *employeeData* class, add the method shown in Listing 7-28.

Listing 7-28. *Modifying the employeeData Class with Insertion Code*

```
 def InsertNewEmployee(self, employeeName):
     dm = dataManager()
     conn = dm.GetConnection()
     comm = dm.GetCommand("INSERT INTO Employee VALUES➥
(1, @name, @birthDate, @hireDate)", conn)
     try:
         conn.Open()
         comm.Parameters.AddWithValue("@name", employeeName)
         comm.Parameters.AddWithValue("@birthDate", DateTime.Now.ToString())
```

```
        comm.Parameters.AddWithValue("@hireDate", DateTime.Now.ToString())
        comm.ExecuteNonQuery()
        conn.Close()
    except:
        conn.Close()
    if (conn.State == ConnectionState.Open):
        conn.Close()
```

Again, it's best practice not to call this directly from the user interface, so open the *employeeBusiness* class and add to it the method shown in Listing 7-29.

Listing 7-29. *Modifying the employeeBusiness Class with Insertion Code*

```
def InsertNewEmployee(self, employeeName):
        emData = employeeData()
        emData.InsertNewEmployee(employeeName)
```

Now we can add an additional text box and button to the form. This set will be for adding an employee to the table. Name the text box *txtNewEmployee* and the button *btnAdd,* with the text label of *Add Name* (Figure 7-10). When you run the application, you should be able to add an employee to the table.



Figure 7-10. *A set of UI controls to add an employee*

■ **Tip** The UI is admittedly quite bland. The key here is the underlying structure, not spending a lot of time on a test UI that you know will not see the light of day. There are times in development when you'll be creating rather sparse UIs for the purpose of quickly testing (sometimes known as *mocking up*) an idea, so don't get too caught up in always trying to make a whiz-bang UI, unless you plan to demo it to customers or your boss.

Deleting an Employee

Lastly, let's wire some code to delete an employee from the table. Add to the *employee-Data* class the code in Listing 7-30.

Listing 7-30. *Adding Deletion Code to the employeeBusiness Class*

```
def DeleteEmployee(self, employeeID):
        dm = dataManager()
        conn = dm.GetConnection()
        comm = dm.GetCommand("DELETE FROM Employee WHERE➥
employeeID = @employeeID", conn)
        try:
            conn.Open()
            comm.Parameters.AddWithValue("@employeeID", employeeName)
            comm.ExecuteNonQuery()
            conn.Close()
        except:
            conn.Close()
        if (conn.State == ConnectionState.Open):
              conn.Close()
```

Now we can call that code from the business layer (Listing 7-31).

Listing 7-31. *Calling the Deletion Code from the Business Layer*

```
    def DeleteEmployee(self, employeeID):
        emData = employeeData()
        emData.DeleteEmployee(employeeID)
```

The final step is to create some UI controls and then to run the application to try it out. For your convenience, Listing 7-32 shows the entire UI code file, Listing 7-33 shows the entire business layer code file, Listing 7-34 shows the entire data manager code file, and Listing 7-35 shows the entire employee data code file.

Listing 7-32. *The Entire UI Code File*

```
import System
from System.Windows.Forms import *
from System.ComponentModel import *
from System.Drawing import *
from clr import *
from employeeBusiness import *
class IPData: # namespace

    class Form1(System.Windows.Forms.Form):
        """type(_txtEmployeeID) == System.Windows.Forms.TextBox, type➥
(txtNewEmployee) == System.Windows.Forms.TextBox, type(_btnAdd) ==➥
System.Windows.Forms.Button, type(_txtDelEmployee) == System.Windows.Forms.TextBox,➥
type(_btnDelete) == System.Windows.Forms.Button, type(_btnSubmit) ==➥
System.Windows.Forms.Button"""
        __slots__ = ['_txtEmployeeID', '_txtNewEmployee', '_btnAdd',➥
'_txtDelEmployee', '_btnDelete', '_btnSubmit']
        def __init__(self):
            self.InitializeComponent()

        @accepts(Self(), bool)
        @returns(None)
        def Dispose(self, disposing):
            super(type(self), self).Dispose(disposing)

        @returns(None)
        def InitializeComponent(self):
            self._txtEmployeeID = System.Windows.Forms.TextBox()
            self._btnSubmit = System.Windows.Forms.Button()
            self._txtNewEmployee = System.Windows.Forms.TextBox()
            self._btnAdd = System.Windows.Forms.Button()
            self._txtDelEmployee = System.Windows.Forms.TextBox()
            self._btnDelete = System.Windows.Forms.Button()
            self.SuspendLayout()
            #
            # txtEmployeeID
            #
            self._txtEmployeeID.Location = System.Drawing.Point(12, 12)
            self._txtEmployeeID.Name = 'txtEmployeeID'
            self._txtEmployeeID.Size = System.Drawing.Size(260, 20)
            self._txtEmployeeID.TabIndex = 0
```

```
#
# btnSubmit
#
self._btnSubmit.Location = System.Drawing.Point(108, 38)
self._btnSubmit.Name = 'btnSubmit'
self._btnSubmit.Size = System.Drawing.Size(75, 23)
self._btnSubmit.TabIndex = 1
self._btnSubmit.Text = 'Get Name'
self._btnSubmit.UseVisualStyleBackColor = True
self._btnSubmit.Click += self._btnSubmit_Click
#
# txtNewEmployee
#
self._txtNewEmployee.Location = System.Drawing.Point(12, 67)
self._txtNewEmployee.Name = 'txtNewEmployee'
self._txtNewEmployee.Size = System.Drawing.Size(260, 20)
self._txtNewEmployee.TabIndex = 2
#
# btnAdd
#
self._btnAdd.Location = System.Drawing.Point(108, 93)
self._btnAdd.Name = 'btnAdd'
self._btnAdd.Size = System.Drawing.Size(75, 23)
self._btnAdd.TabIndex = 3
self._btnAdd.Text = 'Add Name'
self._btnAdd.UseVisualStyleBackColor = True
self._btnAdd.Click += self._btnAdd_Click
#
# txtDelEmployee
#
self._txtDelEmployee.Location = System.Drawing.Point(12, 122)
self._txtDelEmployee.Name = 'txtDelEmployee'
self._txtDelEmployee.Size = System.Drawing.Size(260, 20)
self._txtDelEmployee.TabIndex = 4
#
# btnDelete
#
self._btnDelete.Location = System.Drawing.Point(99, 148)
self._btnDelete.Name = 'btnDelete'
self._btnDelete.Size = System.Drawing.Size(93, 23)
self._btnDelete.TabIndex = 5
self._btnDelete.Text = 'Delete Name'
```

```
            self._btnDelete.UseVisualStyleBackColor = True
            self._btnDelete.Click += self._btnDelete_Click
            #
            # Form1
            #
            self.ClientSize = System.Drawing.Size(284, 264)
            self.Controls.Add(self._btnDelete)
            self.Controls.Add(self._txtDelEmployee)
            self.Controls.Add(self._btnAdd)
            self.Controls.Add(self._txtNewEmployee)
            self.Controls.Add(self._btnSubmit)
            self.Controls.Add(self._txtEmployeeID)
            self.Name = 'Form1'
            self.Text = 'Form1'
            self.Load += self._Form1_Load
            self.ResumeLayout(False)
            self.PerformLayout()

        @accepts(Self(), System.Object, System.EventArgs)
        @returns(None)
        def _Form1_Load(self, sender, e):
            pass

        @accepts(Self(), System.Object, System.EventArgs)
        @returns(None)
        def _btnSubmit_Click(self, sender, e):
            employee = employeeBusiness()
            self._txtEmployeeID.Text =➥
employee.GetEmployeeNameByID(self._txtEmployeeID.Text)

        @accepts(Self(), System.Object, System.EventArgs)
        @returns(None)
        def _btnAdd_Click(self, sender, e):
            employee = employeeBusiness()
            employee.InsertNewEmployee(self._txtNewEmployee.Text)

        @accepts(Self(), System.Object, System.EventArgs)
        @returns(None)
        def _btnDelete_Click(self, sender, e):
            employee = employeeBusiness()
            employee.DeleteEmployee(self._txtDelEmployee.Text)
```

Listing 7-33. *The Entire Business Layer Code File*

```
from employeeData import *

class employeeBusiness:
    "The business class for the Employee table."

    def GetEmployeeNameByID(self, employeeID):
        emData = employeeData()
        return emData.GetEmployeeNameByID(employeeID)

    def InsertNewEmployee(self, employeeName):
        emData = employeeData()
        emData.InsertNewEmployee(employeeName)

    def DeleteEmployee(self, employeeID):
        emData = employeeData()
        emData.DeleteEmployee(employeeID)
```

Listing 7-34. *The Entire Data Manager Code File*

```
import clr
clr.AddReference("System")
clr.AddReference("System.Data")
from System import *
from System.Data import *
from System.Data.SqlClient import *

class dataManager:
    "The data layer manager for our IronPython test application"

    def GetConnection(self):
        "Gets a new SqlConnection object"
        conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=False;➥
Max Pool Size=100;Min Pool Size=5;Pooling=True")
        return conn

    def GetCommand(self, commandString, connection):
        "Gets a new SqlCommand object"
        comm = SqlCommand(commandString, connection)
        return comm
```

Listing 7-35. *The Entire Employee Data Code File*

```
import clr
clr.AddReference("System")
clr.AddReference("System.Data")
from System import *
from System.Data import *
from System.Data.SqlClient import *
from dataManager import *

class employeeData:
    "The data class for the Employee table."

    def GetEmployeeNameByID(self, employeeID):
        result = String.Empty
        dm = dataManager()
        conn = dm.GetConnection()
        comm = dm.GetCommand("GetEmployeeNameByID", conn)
        comm.CommandType = CommandType.StoredProcedure
        try:
            conn.Open()
            comm.Parameters.AddWithValue("@employeeID", employeeID)
            reader = comm.ExecuteReader()
            if (reader.Read()):
              result = reader.GetString(0)
              reader.Close()
              conn.Close()
              return result
        except:
              conn.Close()
        if (conn.State == ConnectionState.Open):
              conn.Close()
              return result


    def InsertNewEmployee(self, employeeName):
        dm = dataManager()
        conn = dm.GetConnection()
        comm = dm.GetCommand("INSERT INTO Employee VALUES (1, @name, @birthDate,➥
@hireDate)", conn)
        try:
            conn.Open()
```

```
            comm.Parameters.AddWithValue("@name", employeeName)
            comm.Parameters.AddWithValue("@birthDate", DateTime.Now.ToString())
            comm.Parameters.AddWithValue("@hireDate", DateTime.Now.ToString())
            comm.ExecuteNonQuery()
            conn.Close()
        except:
            conn.Close()
        if (conn.State == ConnectionState.Open):
            conn.Close()

    def DeleteEmployee(self, employeeID):
        dm = dataManager()
        conn = dm.GetConnection()
        comm = dm.GetCommand("DELETE FROM Employee➥
WHERE employeeID = @employeeID", conn)
        try:
            conn.Open()
            comm.Parameters.AddWithValue("@employeeID", employeeName)
            comm.ExecuteNonQuery()
            conn.Close()
        except:
            conn.Close()
        if (conn.State == ConnectionState.Open):
            conn.Close()
```

Summary

We've covered basic SQL operations and how to connect to SQL Server with IronPython code, we've worked with XML via RSS feeds and created a document from scratch, and we consumed a comma-separated value file. We also looked at what makes a data layer effective and built a simple IronPython data layer to keep our presentation, business, and data code cleanly divided.

CHAPTER 8

Caught in a Web

"I calculated the total time that humans have waited for web pages to load. It cancels out all the productivity gains of the information age. Sometimes I think the web is a big plot to keep people like me away from normal society."

— Scott Adams

It's hard to argue that the invention and spread of the Internet has been anything other than monumental and certainly one of the most important technology developments ever, one that has allowed people to communicate, perform business transactions, and share information faster than anyone could have predicted at its inception. Entire economic cultures have been created and destroyed in the life cycle of the Internet. We'll look at how IronPython lets us build web sites quickly and easily, and along the way you will see how everything we've learned so far applies as effectively to web development as to desktop development.

.NET, IIS, and the Road to Today

In the good old days (if you prefer to wear rose-colored glasses) web pages used to be very static entities. They didn't change much, they offered a terribly limited amount of interaction for the end user, and were essentially, well, ugly. Web pages then and now are created primarily using HTML, which stands for HyperText Markup Language. It's visually very similar to the XML we explored in the last chapter, in that tags are used to describe elements on the page; the difference is that your browser will handle just about any garbage you throw at it, whereas strict XML is much less forgiving. HTML, without any styling applied to it, is also terribly plain (see Figure 8-1, Listing 8-1).



Figure 8-1. *A sample web page, created in HTML and viewed in Firefox*

Listing 8-1. *HTML That Creates the Page in Figure 8-1*

```
<html>
<head>
    <title>A sample HTML page</title>
</head>
<body>
    <h1>This is the title of my page!</h1>
    <h2>It's not terribly interesting.</h2>
    <p>This is the language that is used to construct web pages.</p>
    <!-- This is a comment. You won't see it on the page. -->
</body>
</html>
```

In those same good old days, styling had been applied to the markup directly, in the same file. Although this does allow for some control over the appearance of a page, it creates a maintenance nightmare. Certainly you could create some unique and innovating styles, but even a simple change to the size of a single font on your page would require you to modify the style on each and every page that used it. Like a broken record, I've mentioned many times the importance of keeping code separate; the advent of CSS (Cascading Style Sheets), despite the problems associated with it, is significant in the web development story. CSS allows web developers to separate their markup from their style, allowing them to make site-wide updates to single style sheet files and see the results across all pages that use that style sheet (see Figure 8-2, Listings 8-2 and 8-3).

Tip If for some reason you do not have Internet Information Services (IIS) installed on your computer, I recommend visiting Microsoft's IIS site at `http://www.iis.net` for downloads and installation instructions. It will allow you to work with web pages outside of .NET; otherwise you won't see certain behaviors if you try to browse via the filesystem.



Figure 8-2. *The same web page with a small amount of CSS applied*

Listing 8-2. *HTML Modified with a Link to the Style Sheet*

```
<html>
<head>
    <title>A sample HTML page</title>
    <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <h1>This is the title of my page!</h1>
    <h2>It's not terribly interesting.</h2>
    <p>This is the language that is used to construct web pages.</p>
    <!-- This is a comment. You won't see it on the page. -->
</body>
</html>
```

Listing 8-3. *The CSS for the Web Page We Created*

```
/* styles.css */
body { font-family: calibri; }
h1 { text-decoration: underline; font-style: italic; }
h2 { float: right; font-family: verdana; }
p { margin-left: 35px; border: 1px solid #000; width: 380px; }
```

■ **Note** What problems exist with CSS? For the longest time, Microsoft and Netscape were engaged in a browser war. Each provider tried to offer features the other didn't, and as a result proprietary methods and code were produced that worked in one but not in the other. To boil down 10+ years of history, CSS is one of the casualties of this war. As of this writing, no browser on the market currently supports the CSS standard precisely, Internet Explorer (IE) has Microsoft-supported conditional markup statements so that you can write CSS that displays in IE but not in Firefox or Opera, and so on.

These are not fringe problems you'll never encounter as a developer. These are everyday, "in the trenches" problems about which people have written entire books. If you need proof that the effects still linger, then head to `http://www.htaccesstools.com/browser-check/` and see what information your browser is sending to web servers. A perfect example is Internet Explorer 7; if you browse this page with IE7, you'll see that the user-agent is reported as "Mozilla/4.0 (compatible; MSIE 7.0;)" along with some other data unique to your system configuration. This is a direct result of the browser wars, when one browser would represent itself as another. It's indeed a tangled web out there.

Web pages are delivered to a user after being served up by web servers; the user's browser makes a specially formed request to the server for a specific piece of content, and the server renders that content and sends the response back to the user. Microsoft's flagship web server technology is the Internet Information Services platform, or IIS. Consequently, Microsoft has made a significant effort to integrate the .NET framework with IIS, which has resulted in quite the powerful entity for developers. Because Microsoft has applied their .NET technology to their IIS server, you don't have to throw your desktop knowledge out the window; desktop and web development have become not-so-distant cousins.

In recent years, there has been a significant push to make web applications look and feel more like desktop applications. Google Mail, MapQuest, YouTube, MySpace, and Facebook are just a few examples of advanced web applications that provide functionality that is leaps and bounds above the Internet of just a few years ago. They are a mix of **server-side** and **client-side code** and effectively blur the line between desktop and web software. The .NET framework, in combination with IronPython and IIS, can be used to create powerful and advanced web applications that behave like desktop ones, and in some cases accomplish tasks that would be much more difficult for a desktop application to perform. They also have some unique benefits over traditional desktop applications that make maintenance a breeze.

■Note *Server-side code* is code that is executed entirely on the server itself. The IronPython code you write will be executed on the server by IIS; the *results* of that execution are what is sent back to the user. *Client-side code* is the opposite; it is code that is executed entirely by the client. JavaScript is the most well-known client-side code. JavaScript is executed by the user's browser and actually suffers from some of the browser war effects that plague CSS, namely, different browsers supporting custom methods or pieces of the JavaScript language.

If you've spent any time in the Windows environment, no doubt you've seen the Automatic Updates that pop up from time to time. What's happening is that Microsoft developers are patching code for the Windows operating system, but the only way for you to benefit from those patches is to get the code on your machine and to apply it to the files you already have. Nowadays things are pretty advanced and this all generally happens in the background for you. The issue still remains that when the developers at Microsoft want to fix an issue or provide some new feature, *additional steps must be taken for end users to benefit from them*. This is where web development can truly shine: if I make a change to the code or markup of files in my web application, those changes are immediately visible to the end user and don't require additional work on their part. This should stand out as a big deal.

■Note If you count yourself among the "grizzled ancient" developers, no doubt you've encountered the dreaded DLL Hell. To the lucky ones who aren't familiar with this concept, it happens when libraries are incompatible with one another, resulting in error messages, unpredictable behavior, or a complete lack of functionality for your end users once things are out of sync. It can be a truly awful situation to diagnose when things get complicated. Web development can skirt around that issue entirely, because the only thing users can browse to is what you've provided them, so everyone should be on the same page at all times. In fact, this problem was a big motivator for the creation of .NET assemblies.

.ASPX and You

Web pages have generally had the file extension of .htm or .html; you may also have seen .php, .asp, .aspx, or a variety of other extensions. In the .NET world, pages with .NET code running behind them generally have the extension of .aspx, which is a nod to the older style of Microsoft web page called ASP, or Active Server Pages.

When creating web sites in Visual Studio, the process is very similar to creating Forms applications. You will still have a solution that can contain multiple projects, you will still have .py source code files, and so on. Most of your knowledge transfers smoothly right into web development, but with some additional tidbits added on. Probably the best way

to jump into this is at the deep end, so the first thing we'll do is acquire the sample project from Microsoft and build from there.

■Note At the time of my writing this, there was no set of project templates for IronPython that can be used in Visual Studio 2008. The CodePlex site indicates that these templates are forthcoming with future releases. But for the time being we'll have to take a few steps manually. For the sake of simplicity, we'll use the sample IronPython for ASP .NET project Microsoft provides; it's a fine starting point for web development in IronPython.

You can download the sample project from `http://aspnet.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=17613`. You'll want to download and unzip the "ASP .NET WebForms IronPython Sample" to your desktop or any convenient location. Open Visual Studio, and then click File, Open, and Web Site. Navigate to the folder where you unzipped the project files and open the "ironpython-webform-sample" site (see Figure 8-3).

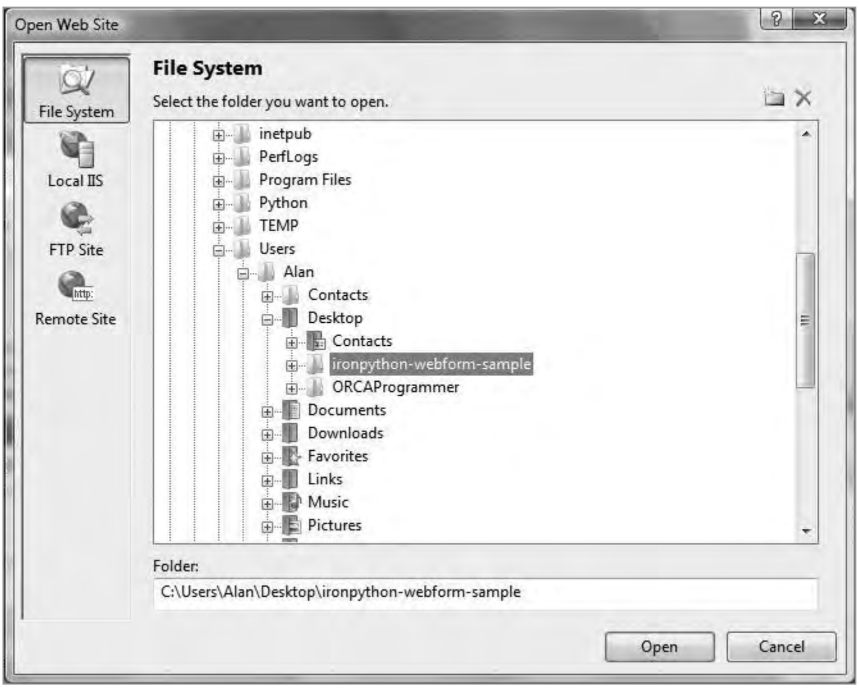


Figure 8-3. *Opening a web site in Visual Studio*

After opening the project, you'll be presented with a screen similar to the one in Figure 8-4. In the Solution Explorer on the right, notice that a variety of folders and files are already present. Double-click on *Default.aspx* to open the file. Compare the markup in this file to the markup in the HTML file we created earlier in the chapter. What sorts of things do you notice that are different?

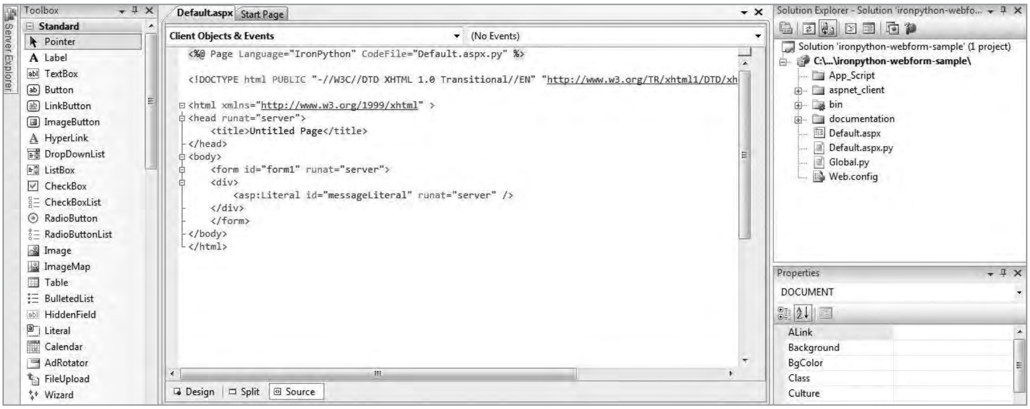


Figure 8-4. *The project as it initially appears, with Default.aspx opened*

The two things I'd like to point out in particular are the `<%@ Page … %>` line at the beginning of the file and the `<asp:Literal />` line that appears in between the `<div>` tags. These lines are specific to .NET and reveal a bit about what's happening behind the scenes. The exact text of the *Page* directive at the top of the file is as shown in Listing 8-4.

Listing 8-4. *The Page Directive from Default.aspx*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" %>
```

You won't see this line of text in the source code of the page once it's rendered, nor will you see it physically on the page. This is a bit of "inline" code that is solely for the server to know about and operate on. It tells the server two things: (1) that any server-side code in this page should be processed as IronPython (as opposed to C#, VB .NET, or other .NET languages) and (2) that the IronPython code for this file can be found in *Default.aspx.py*.

■**Note** Code-behind files are the method of developing code that I will tout to you. You can actually write all your IronPython code using what's called the *inline method*, meaning the IronPython code and the markup are stored in the same file, with the IronPython code stored in between specially structured <script> tags. Personally, I hate this method of development for most situations. I prefer to keep my markup and my code separate. Be aware that there is another way, but I wouldn't consider it the better way and I'll be using the code-behind style. It's also worth noting that Microsoft chose to structure their example this way as well.

Every *.aspx* page will have at least one line in `<% %>` brackets, and that line defines the *Page* element, the language, and so on. Without that, you can't have a functional *.aspx* page. When I refer to code-behind versus inline, I am referring to the act of writing your IronPython code directly in the page versus in a separate *.py* file on the server. You can mix and match, and in some cases this is an appropriate methodology, but I would highly recommend the code-behind model wherever possible.

The line with the *Literal* control is shown in Listing 8-5. This has some special syntax parameters based on the type of object that it is, which we'll cover next.

Listing 8-5. *The Markup That Declares the Literal Control*

```
<asp:Literal ID="messageLiteral" runat="server" />
```

The tag looks something like a regular HTML tag, but with some extra information. First, it's prefixed with *asp*. This denotes an ASP .NET server control, which is distinct from an HTML tag; it provides additional functionality behind the scenes, in addition to hooks for style information and other features. Note that for this to work properly, you *must* include `runat="server"` in the tag. If this property is not set correctly, then the code will not even run.

We've also defined an *ID* property and set it to `"messageLiteral"`. This uniquely identifies the control on the page, to avoid conflict with any other control. Thus, if you have a control with its *ID* property set to *foo*, you cannot have another control on the same page with an ID of *foo*.

Now is a fine time to introduce you to an aspect of .NET web development that can trip people up at the outset. Press F5 to run the web site. If you see a notification that debugging is not enabled, feel free to select the option that allows Visual Studio to enable debugging in the *web.config* file. The output of the page should look like Figure 8-5).

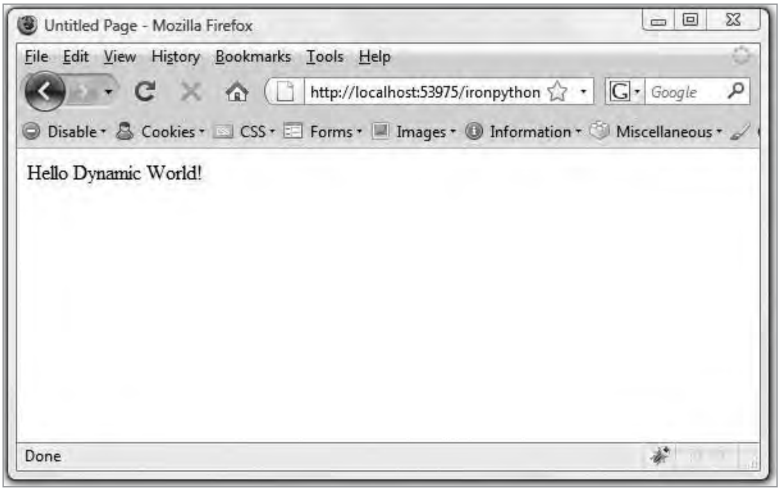


Figure 8-5. *The output from the IronPython web site application*

Note I'll be using Mozilla Firefox as my browser of choice in my examples. Unless otherwise noted, it won't matter if you're using Internet Explorer, Opera, or some other browser. You will just see a slightly different browser interface than in my example figures.

This text was not in the markup of the page. We were just looking at it, and the only things present were some structural tags (such as *divs*) and a *Literal* control. How is the text "Hello Dynamic World!" now present? View the source to this page. In Firefox you can press Ctrl+U; in Internet Explorer you can right-click on the page and click View Source. Regardless of your browser, you should see markup very similar to that in Listing 8-6.

Listing 8-6. *The Markup the Browser Is Rendering*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head><title>
            Untitled Page
</title></head>
```

```
<body>
    <form name="form1" method="post" action="default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value=➥
"/wEPDwUBMA9kFgICAw9kFgICAQ8WAh4EVGV4dAUUSGVsbG8gRHluYW1pYyBXb3JsZCF➥
kZC6yi9dXNE5UaiAWKDjOWcQXdRWj" />
</div>

    <div>
            Hello Dynamic World!
    </div>
    </form>
</body>
</html>
```

■Note Your ViewState likely can and will be different from mine. This is no cause for concern.

This is actually quite a bit different! The *Literal* control has been replaced with the text “Hello Dynamic World!”, there’s a hidden input tag called "`__VIEWSTATE`" that has a bunch of crazy stuff in it, and the *form* tag now has a method and an *action* property. What happened?

Here’s where the server is stepping in and working its magic. Before we delve into the ViewState business, let’s end our debugging session by closing the browser and returning to Visual Studio. Open *Default.aspx.py*; the code should look like Listing 8-7.

Listing 8-7. *The Code from Default.aspx.py*

```
def Page_Load(sender, e):
    messageLiteral.Text = "Hello Dynamic World!"
    pass
```

Aha! Some IronPython code is running back here. The ASP .NET page life cycle defines a variety of events that happen at different times; there is some code being executed during the `Page_Load` event that changes the *Text* property of the *messageLiteral* control to say “Hello Dynamic World!”. This is processed by the server. The server renders the appropriate text to the end user, and the markup is therefore totally different for the end user than for the developer. This makes sense, but we need to point it out explicitly: the markup of an ASP .NET web page frequently comprises controls that technically function as placeholder elements. The IronPython code you write can dramatically and completely change what the end user sees to provide the functionality you need.

Let's modify the *messageLiteral.Text* property with our own text and see what happens. Once you've changed the code to match Listing 8-8, run the application again (Figure 8-6).

Listing 8-8. *The Code from Default.aspx.py Updated*

```
def Page_Load(sender, e):
    messageLiteral.Text = "We made some changes to the code-behind file; now➥
this output is TOTALLY DIFFERENT!"
    pass
```

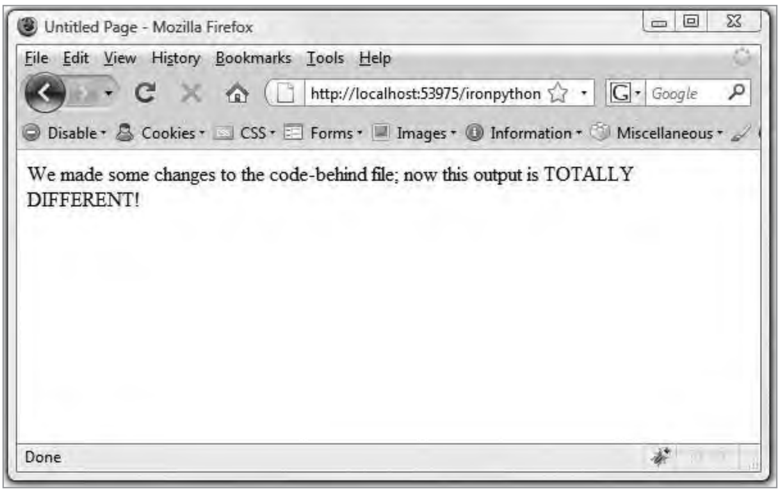


Figure 8-6. *The code-behind file has been modified.*

The State of the View

Web pages are, by definition, stateless. This means that no information is stored between requests to the server. ViewState is designed to get around that issue and to preserve the state of the page between requests. Listing 8-9 shows what my ViewState looks like from the page we rendered.

Listing 8-9. *The ViewState Tag from the Page in Figure 8-5*

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value=➥
"/wEPDwUBMA9kFgICAw9kFgICAQ8WAh4EVGV4dAUUSGVsbG8gRHluYW1pYyBXb3JsZCF➥
kZC6yi9dXNE5UaiAWKDjOWcQXdRWj" />
```

This tag is generated automatically by .NET and is a hidden input on the page itself. You won't see it while looking at the page, but you can view it in the page source. You are also permitted to store and retrieve information from the ViewState if you need to persist information between requests.

Caution Although the state of the page controls looks to be encrypted, it is not; it is merely *encoded*. That means anyone can actually decode this information by using a surprisingly small amount of custom code or one of a variety of tools that already exist for decoding ViewState and reading its contents. *DO NOT STORE SENSITIVE OR PRIVATE INFORMATION IN VIEWSTATE*. I cannot emphasize this enough. If you wouldn't want it displayed on the page in gigantic, blinking red letters for the whole world to write down and remember forever, don't store it in ViewState. It's not a secured method of storing information.

Let's try storing a little text in the ViewState and see what happens to the page markup (Listing 8-10, Figure 8-7, Listing 8-11).

Listing 8-10. *Modifying the Page_Load Event to Store Text in the ViewState*

```
def Page_Load(sender, e):
    messageLiteral.Text = "We made some changes to the code-behind file; now this➥
output is TOTALLY DIFFERENT!"
    if (ViewState["test"] == None):
            ViewState["test"] = "Adding to ViewState!"
    pass
```

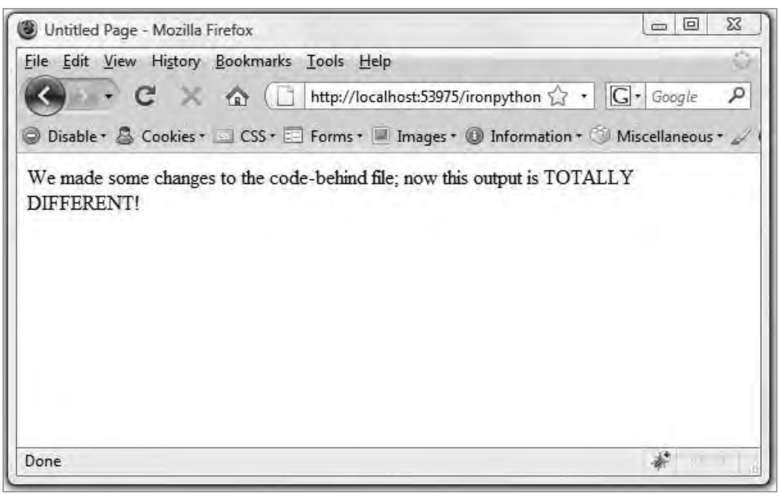


Figure 8-7. *The output is the same, but with additional information in the ViewState tag.*

Listing 8-11. *Comparing the Old ViewState to the New One*

```
Original ViewState:
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value=➥
"/wEPDwUBMA9kFgICAw9kFgICAQ8WAh4EVGV4dAUUSGVsbG8gRHluYW1pYyBXb3JsZCF➥
kZC6yi9dXNE5UaiAWKDjOWcQXdRWj" />

New ViewState:
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value=➥
"/wEPDwUBMA8WAh4EdGVzdAUUQWRkaW5nIHRvIFZpZXdTdGF0ZSEWAgIDD2QWAgIBDx➥
YCHgRUZXh0BVNXZSBtYWRlIHNvbWUgY2hhbmdlcyB0byB0aGUgY29kZS1iZWhpbmQgZml➥
sZTsgbm93IHRoaXMgb3V0cHV0IGlzIFRPVEFMTFkgREIGRkVSRU5UIWRk5OjvdUOw5lXXT7➥
JLm7qYvLfariA=" />
```

The ViewState is now a bit larger due to the encoding of the characters that we added. How can we retrieve that information and use it? Luckily, getting at the data is a trivial matter. We'll get the data from ViewState and append it to the *Literal* tag (Listing 8-12, Figure 8-8).

Listing 8-12. *Retrieving Data from the ViewState Tag*

```
def Page_Load(sender, e):
    messageLiteral.Text = "We made some changes to the code-behind file; now this➥
output is TOTALLY DIFFERENT!"
    if (ViewState["test"] == None):
        ViewState["test"] = "Adding to ViewState!"
        messageLiteral.Text += " - The ViewState additionally contains " +➥
ViewState["test"]
    pass
```

Tip Storing information in ViewState is an exercise in restraint and judgment. In addition to the fact that encoded ViewState data increases the page size for the end user (and therefore incurs an additional bandwidth increase), if you store more complex business objects in the ViewState you will find a performance cost in storage and retrieval as the object is encoded and decoded. Use it sparingly and only when necessary, with a preference toward simpler objects when possible.

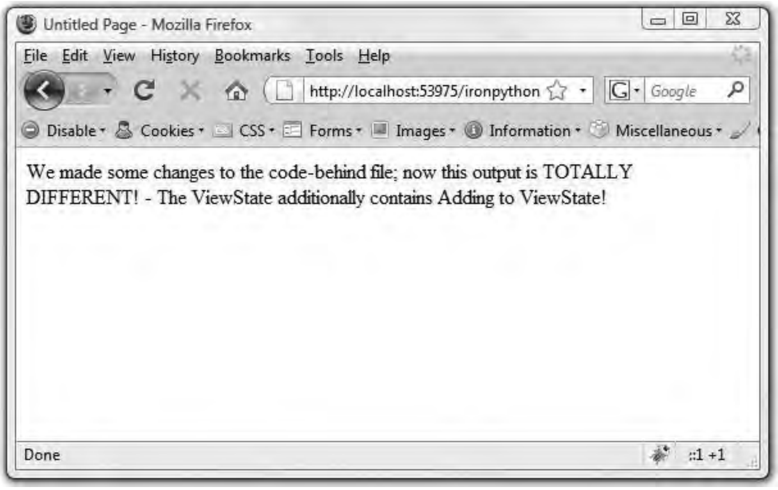


Figure 8-8. *The data we store in the ViewState tag is easily retrieved.*

POST

One of the bits of code that .NET added to the page was the additional set of parameters in the form tag (Listing 8-13). These parameters serve two functions: the first is dictating the type of request that will be made to the server, and the second is defining which page we are sending the request to. You can see this by viewing the source of the web page in your browser.

Listing 8-13. *The Modified form Tag*

```
<form name="form1" method="post" action="default.aspx" id="form1">
```

The act of sending the request back to the current page is known as the *PostBack*. In the PostBack, an HTTP request using the verb *POST* is made. This process is the backbone of .NET web development, as we'll see shortly.

■ **Note** There are a variety of HTTP verbs, including POST, GET, PUT, HEAD, and many more. They serve a variety of different purposes; the two you'll deal with most frequently are GET and POST. The most obvious difference between GET and POST is that a GET request has all the form data encoded into the address bar URL, whereas a POST contains all form data in the message body and is theoretically the more secure method. Also note that POSTs frequently cannot be cached, whereas GETs can, which can influence performance considerations.

Creating a Simple Form

We can create a basic form very simply in IronPython and ASP .NET. We'll create a simple form that takes some information from the user; then we'll submit that information via a POST request and discuss some of the benefits and limitations of this development model.

Our form will serve as a way for users to create accounts for a fake online application. We'll create some basic introductory text and markup and see what happens during the ASP .NET page life cycle. Modify the *Default.aspx* markup so that it looks like Listing 8-14. We'll also need to remove some of the existing code from *Default.aspx.py* so that everything runs (Listing 8-15).

Listing 8-14. *The Create Account Page Markup for Default.aspx*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Create an Account</title>
</head>
<body>
    <form id="form1" runat="server">
    <div id="intro">
    <h1>Create an Account</h1>
    <p>To create an account for this fictional system, please fill out the fields➥
below and click Submit.</p>
    </div>
    <div id="userInputForm">
        <table>
            <tr>
                <td>
                    Username
                </td>
                <td>
                    <asp:TextBox ID="txtName" runat="server"></asp:TextBox>
                </td>
            </tr>
```

```
            <tr>
                <td>
                    Password
                </td>
                <td>
                    <asp:TextBox ID="txtPassword" runat="server"➥
TextMode="Password"></asp:TextBox>
                </td>
            </tr>
        </table>
        <asp:Button ID="btnSubmit" runat="server" Text="Submit" /><br/>
        <asp:Literal ID="litFeedback" runat="server" />
    </div>
    </form>
</body>
</html>
```

Listing 8-15. *The Modified Default.aspx.py File*

```
def Page_Load(sender, e):
    pass
```

Running the application should produce a page that looks something like Figure 8-9. You can fill in the fields with some sample data; then click the Submit button. You should see the page "flash"; then it will return to the same location. This is the PostBack in action. The browser is sending information to the server; the server processes and executes any server-side code and then sends the response back. In our case, we haven't created any code to execute on PostBack, so it looks like nothing happened (although it's likely that the password field is blank now, which is a by-design security feature of modern browsers, to prevent people from copying and pasting).

So what exactly is happening when the browser POSTs to the server? The terrific Firefox add-on called Firebug can help developers examine what's happening at a low level during web development. The screenshot in Figure 8-10 shows what is transmitted to the server during the PostBack.

■ **Tip** If you're interested in acquiring Firebug for your own development purposes, you can find it at `http://getfirebug.com/`. It's a totally free add-on to Firefox and contains a variety of useful features for web developers.



Figure 8-9. *The sample Create Account page*

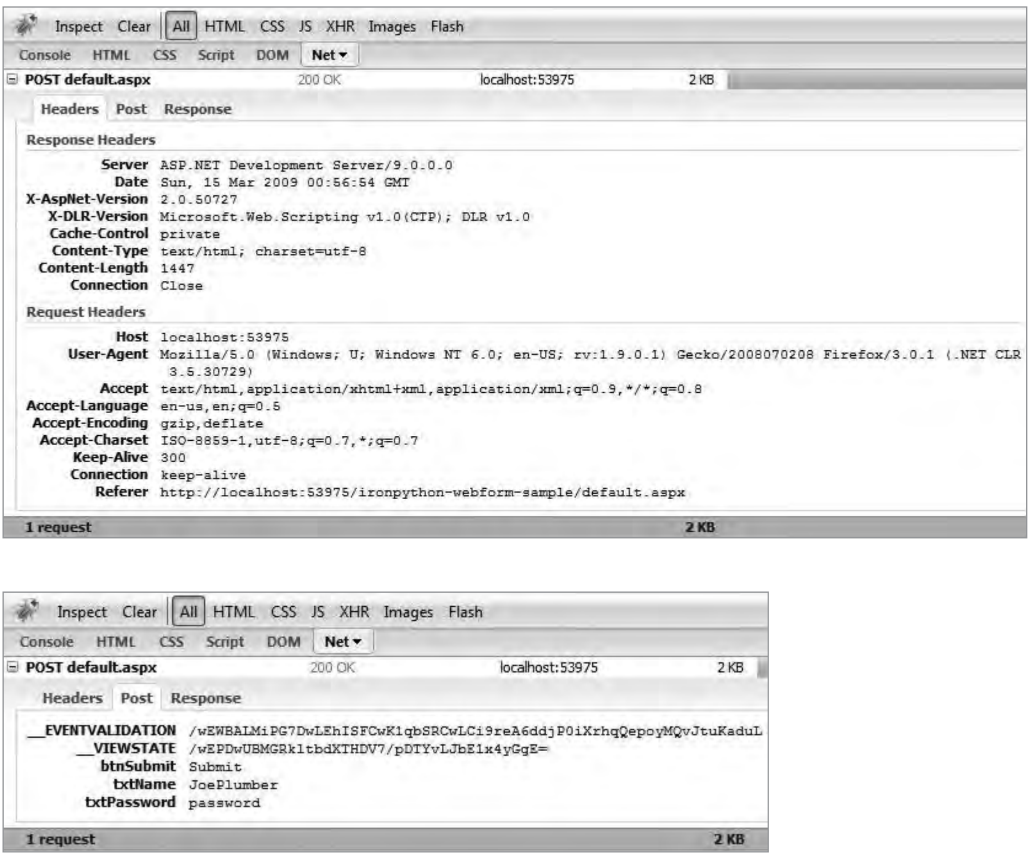


Figure 8-10. *The contents of the POST to the server*

Tip Got those eagle eyes on? You may notice the X-DLR-Version server variable in Figure 8-10. That's new to IronPython for ASP .NET; it holds the version of the current Dynamic Language Runtime that's being used. In my case, I'm using a Community Technology Preview (CTP) of version 1.0.

It's plainly visible that the request is being made to our *Default.aspx* page. Now we can perform a task based on the results of that POST. Open the *Default.aspx.py* file and modify it to look like Listing 8-16.

Listing 8-16. *The Modified Default.aspx.py File*

```
def Page_Load(sender, e):
    if (Page.IsPostBack):
         litFeedback.Text = "You POSTed back: " + txtName.Text + " - " +➥
txtPassword.Text
    pass
```

Now we can run the application again, fill in some sample data (Figure 8-11), and click the Submit button.

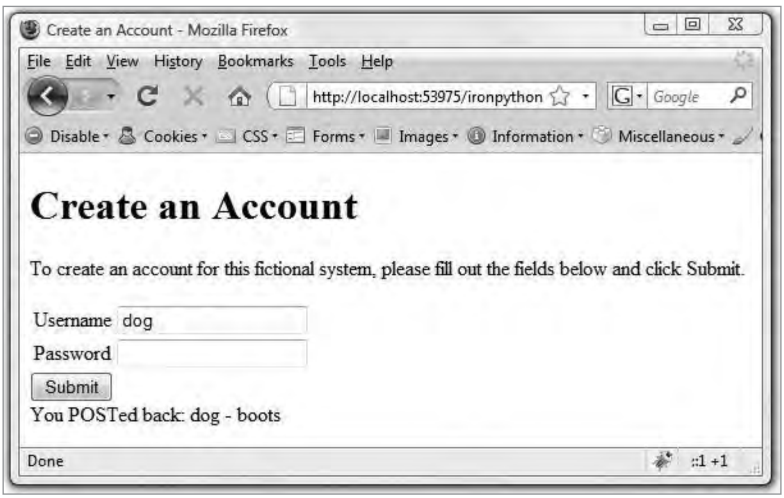


Figure 8-11. *We submitted "dog" and "boots" as our POST data.*

Know Your Limitations

Though effective, this development model is not without its limitations. A perfect example of this can be demonstrated with the form we've created. What happens if we want our form submission to end with our arriving at a Thank You page, which is very common on the web (a little courtesy goes a long way, after all). First, let's add a file called *thankYou.htm* to the project by right-clicking on the solution and clicking Add new file. Modify it so that its markup looks like Listing 8-17.

Listing 8-17. *The Markup for the thankYou.htm Page*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
</head>
<body>
<h1>Thank You!</h1>
<p>We have created your account; you may now log in using the information you➥
supplied during account creation.</p>
</body>
</html>
```

Now we can modify the *Default.aspx.py* file to use this page after submission (Listing 8-18).

Listing 8-18. *Modifying Default.aspx.py to Use the New thankYou.htm Page*

```
def Page_Load(sender, e):
    if (Page.IsPostBack):
           Response.Redirect("~/thankYou.htm")
    pass
```

■ **Note** *Response.Redirect* is a very common method that developers employ to send users along to a particular page, but it's not really perfect for all occasions, as we'll see in just a moment.

If you run the application again, fill out the form, and then click the Submit button, you should find yourself taken to the *thankYou.htm* page (Figure 8-12). In theory, this is great. We filled out the form and got a pleasant thank you for the effort. But how did we get there?



Figure 8-12. *The code successfully redirected us to the thankYou.htm page, but perhaps there's a better way.*

Opening Firebug and examining the request reveals the story (Figure 8-13).

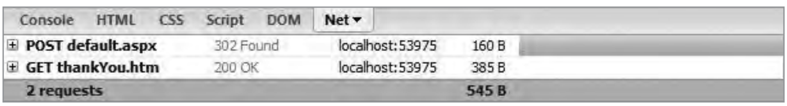


Figure 8-13. *We submitted our request to Default.aspx before hitting the thankYou.htm page.*

Cross-Page PostBacks

There's a method for performing what is known as *cross-page PostBacks* in .NET 2.0 and up. Older versions, such as 1.0 and 1.1, don't allow these operations to be performed, but we're living in luxurious times. However, this luxurious method has a limitation of its own, which we'll discover shortly.

Telling .NET that you want to perform a PostBack to a different URL is very easy. Open *Default.aspx* and modify the button markup to look like Listing 8-19.

Listing 8-19. *Modifying Default.aspx to Use a Different PostBack URL on Submission*

```
<asp:Button ID="btnSubmit" runat="server" Text="Submit"➥
PostBackUrl="~/thankYou.html" />
```

The *PostBackUrl* property tells the server the location we want to POST to instead of the current page, which is the default. Run the application again, fill in the form, and click Submit (Figure 8-14). What happened?

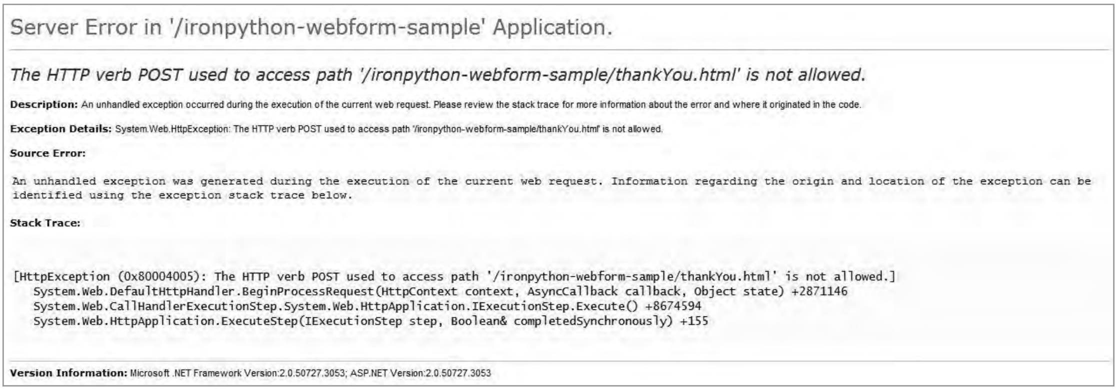


Figure 8-14. *The limitation of cross-page PostBacks revealed*

Here's the wall, although admittedly a minor one. If we want to perform a cross-page PostBack, we have to do it to an ASP .NET page. We can easily modify our *thankYou* page to allow a PostBack. First, rename the file from *thankYou.htm* to *thankYou.aspx*. Visual Studio will inform you that by changing the extension of the file, the file may become unusable. Go ahead and proceed with renaming it. Now modify the markup of *thankYou.aspx* to look like Listing 8-20.

Listing 8-20. *Modifying thankYou.aspx to Accept PostBacks*

```
<%@ Page Language="IronPython" CodeFile="thankYou.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Thank You!</title>
</head>
```

```
<body>
<h1>Thank You!</h1>
<p>We have created your account; you may now log in using the information you➥
supplied during account creation.</p>
</body>
</html>
```

The last step is to create the *thankYou.aspx.py* file and supply the necessary code, which you'll find in Listing 8-21.

Listing 8-21. *The Code for thankYou.aspx.py*

```
def Page_Load(sender, e):
    pass
```

■ **Note** Don't forget to update the *PostBackUrl* in *Default.aspx* as well; we're not POSTing to *~/thankYou.htm* anymore. Now it will be *~/thankYou.aspx*.

You should now be able to run the application, fill out the form, and submit to *thankYou.aspx*. You've created an *.aspx* page from scratch as well as the code-behind file. We're having to take more manual steps than normal because there are no Microsoft-supplied project templates at the moment, but rest assured that those templates are coming (possibly even by the time this book reaches you) and the task will become much simpler.

Let's confirm with Firebug that our work was successful (Figure 8-15).

We've now been able to POST to a different page than the one we're currently on, which eliminates an unnecessary call and reduces the chattiness of our application. So, with that in mind, how do we access data from the previous page? Perhaps we want to display it the way we did before. Is it as simple as that?

■ **Tip** These are critical issues in the realm of search engine optimization (SEO). Search engines see your pages in a much different way than you do, and they place emphasis on things that aren't so obvious to the end user. Although a full discussion of SEO and how to apply those principles to .NET is a topic that could occupy a complete book by itself, it's worth mentioning that by providing the most direct, logical route, we are improving the way a search engine views and indexes our pages. It's never a bad idea to remove needless redirects. Keep it simple!

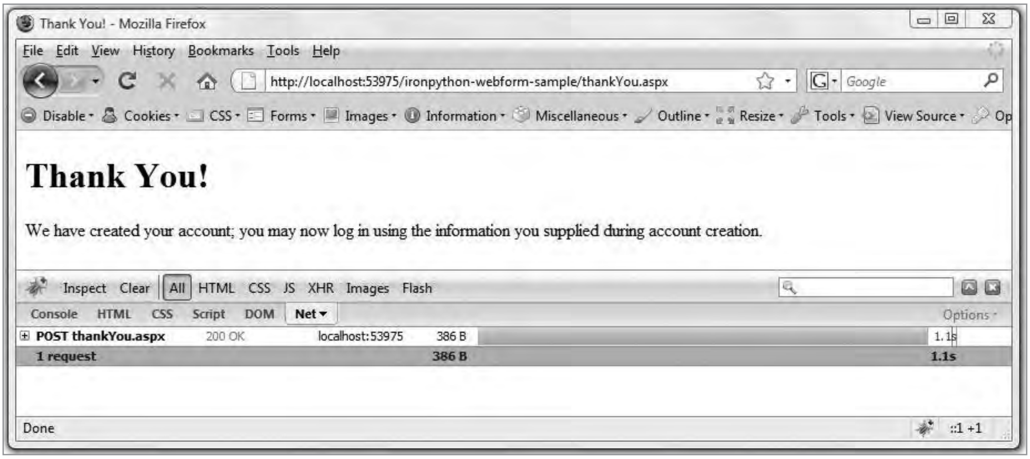


Figure 8-15. *This is much cleaner, from a traffic perspective.*

Accessing Cross-Page Data

Unfortunately, the answer to our preceding question is no, it's not as simple to display data from the previous page as it was before, because we are now on a different page. The good news is that although it's not as simple, it's also not terribly complex. First, we can add a *Literal* control to the *thankYou.aspx* page to hold the results of our cross-page PostBack (Listing 8-22).

Listing 8-22. *The New Markup for thankYou.aspx*

```
<%@ Page Language="IronPython" CodeFile="thankYou.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Thank You!</title>
</head>
<body>
<h1>Thank You!</h1>
<p>We have created your account; you may now log in using the information you➥
supplied during account creation.</p>
<asp:Literal ID="litFeedback" runat="server" />
</body>
</html>
```

Now we can use the *PreviousPage* method to gain access to a control from the page that's submitting a request (Listing 8-23).

Listing 8-23. *Accessing Data from the Previous Page in thankYou.aspx.py*

```
def Page_Load(sender, e):
    prevTxtName = PreviousPage.FindControl("txtName")
    prevTxtPassword = PreviousPage.FindControl("txtPassword")
    litFeedback.Text = prevTxtName.Text + prevTxtPassword.Text
    pass
```

Right-click on *Default.aspx* in the Solution Explorer and select "Set as start page." When you press F5, that page will be the one that appears in your browser. If you run the application now and proceed through submission, you should be presented with a *thankYou.aspx* page that looks something like Figure 8-16.



Figure 8-16. *My values made the trip across pages.*

Validation (for a Reasonable Fee)

You may have caught on to the dirty little secret of this application. It's a terrible offense that more applications commit than you may realize. The offense is one of not validating user input; the application won't fail to function as is if the user doesn't provide any

input, but that doesn't mean that validation of some type is not required. Validation typically includes at least the following tasks.

1. Did the user enter data in all of the required fields?
2. Does the data for each field match any requirements we have for it? (This is a sanity check. For example, did the user enter a phone number for her name, or an e-mail address for a ZIP code?)
3. Have we ensured that the data is safe? What happens if we display it to the screen or save it to the database? Have we protected ourselves against security flaws?

We ourselves can write custom code to perform each of these tasks. Unfortunately, some of them are actually quite complicated, despite how simple they seem. The .NET framework does come with some built-in security features; let's apply a few of them to our application and see how things shape up.

Using the *RequiredFieldValidator*

One of the easiest-to-use validators that .NET provides is the *RequiredFieldValidator*. You can add this control to your page and connect it to a specific control; if, when the user attempts to POST the page, there is no input in the control the *RequiredFieldValidator* is monitoring, the page will not POST. An error message of your choosing will then display, and the user will remain on the current page.

Open the *Default.aspx* file and add the bold blocks of markup where indicated in Listing 8-24; then run the application.

Listing 8-24. *An Improved Default.aspx page with Some RequiredFieldValidators Added*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Create an Account</title>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div id="intro">
        <h1>
            Create an Account</h1>
        <p>
            To create an account for this fictional system, please fill out the➥
fields below
            and click Submit.</p>
    </div>
    <div id="userInputForm">
        <table>
            <tr>
                <td>
                    Username
                </td>
                <td>
                    <asp:TextBox ID="txtName" runat="server"></asp:TextBox>
                </td>
                <td>
                    <asp:RequiredFieldValidator ID="txtNameRequired"➥
runat="server" ControlToValidate="txtName"
                        ErrorMessage="You must provide a ➥
username."></asp:RequiredFieldValidator>
                </td>
            </tr>
            <tr>
                <td>
                    Password
                </td>
                <td>
                    <asp:TextBox ID="txtPassword" runat="server"➥
TextMode="Password"></asp:TextBox>
                </td>
                <td>
                    <asp:RequiredFieldValidator ID="txtPasswordRequired"➥
runat="server" ControlToValidate="txtPassword"
                        ErrorMessage="You must provide a➥
password."></asp:RequiredFieldValidator>
                </td>
            </tr>
        </table>
```

```
        <asp:Button ID="btnSubmit" runat="server" Text="Submit"➥
PostBackUrl="~/thankYou.aspx" /><br />
        <asp:Literal ID="litFeedback" runat="server" />
    </div>
    </form>
</body>
</html>
```

Try to click the Submit button *without* entering any text for the username or password boxes. You should immediately see the error messages appear to the right of the boxes (Figure 8-17); note that the page did not PostBack to itself. The submission process was entirely interrupted by the validators.

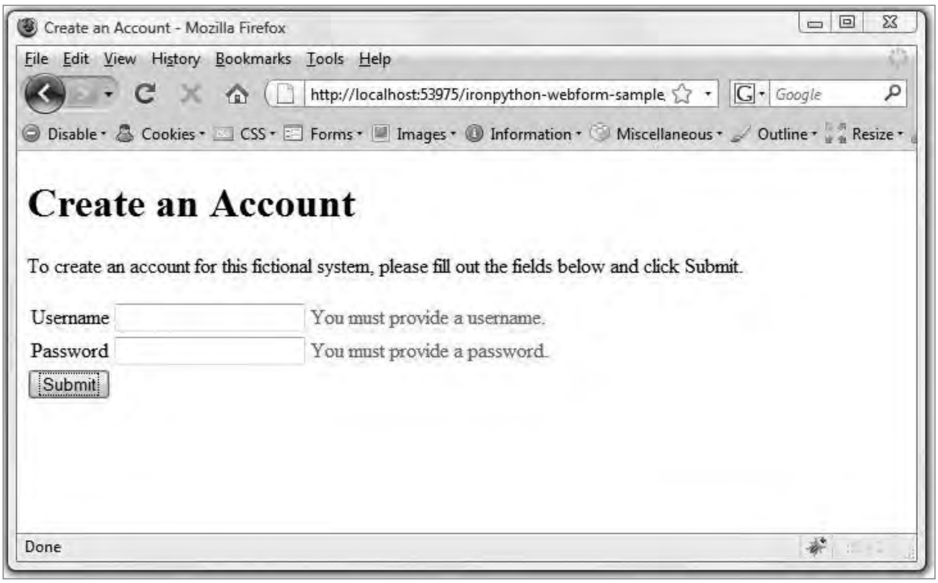


Figure 8-17. *The RequiredFieldValidators have been triggered.*

To use the *RequiredFieldValidator*, you must supply the name of a control to validate; you must also use one *RequiredFieldValidator* per field you wish to validate. You cannot apply one *RequiredFieldValidator* across multiple controls.

■ **Tip** You may have noticed that the page doesn't "flash" when you click the Submit button; the error message appears immediately. This is accomplished through some client-side JavaScript that .NET injects into the page for you when the markup is created and sent to the user. What happens if the user has not enabled JavaScript or is using a browser that doesn't support JavaScript? When the user clicks Submit, the page will appear to PostBack but will refresh to show the error messages. This is a great example of the .NET framework saving you a little legwork.

Handling Errors

So far, our application has been simple enough that we haven't encountered much in the way of errors. Even before we put *RequiredFieldValidators* on the page, the lack of any user input still didn't really disrupt the flow of the application. We can artificially induce an error and see what the user would see, and this alone should prove sufficient evidence of the need for judicious error handling and reporting.

Open *thankYou.aspx.py* and modify it to look like Listing 8-25.

Listing 8-25. *Artificially Causing the Page to Fail—Division by Zero Triggers an Error Automatically.*

```
def Page_Load(sender, e):
    prevTxtName = PreviousPage.FindControl("txtName")
    prevTxtPassword = PreviousPage.FindControl("txtPassword")
    litFeedback.Text = prevTxtName.Text + prevTxtPassword.Text
    impossibleValue = 1 / 0
    pass
```

■ **Note** Why does `1 / 0` trigger an error? The short answer: because of the mathematical axiom $x(y/x) = y$; for example, $2(3/2) = 3$. However, this equation doesn't work with 0. It would yield $0(1/0) = 1$, which is not true. Therefore, the answer is undefined and triggers an exception. There is no value for x such that $x * 0 = 1$.

Run the application and submit some values to the *Default.aspx* page. When it redirects to the Thank You page, you will be returned to Visual Studio and the `impossibleValue = 1 / 0` line will be highlighted. The IDE will inform you that you cannot perform that operation. Close that box by clicking the X in the upper right corner.

Press F5 to continue running the application and take a look at what appears in the browser (Figure 8-18). This is what your end user would see if this were a live, production application.

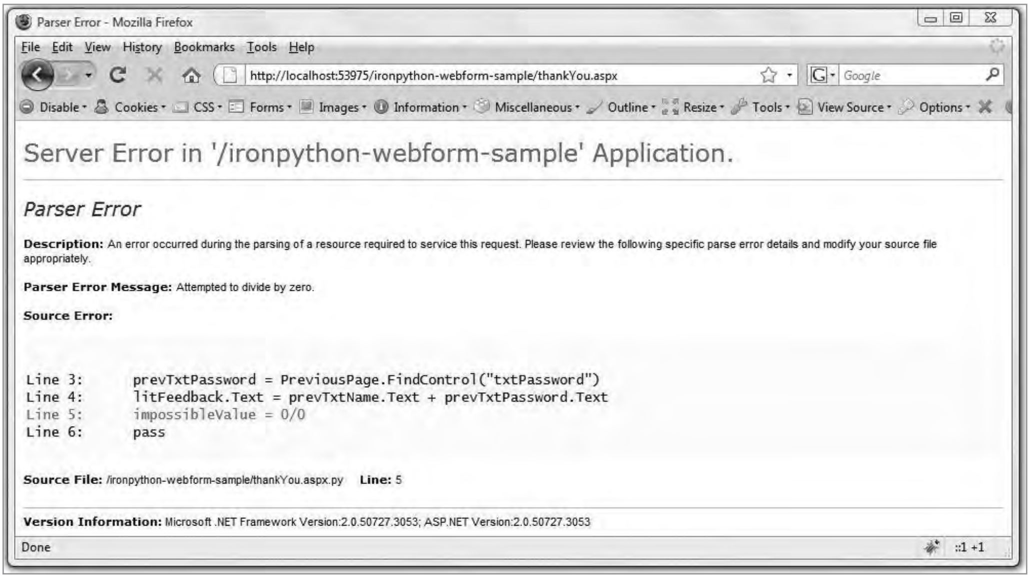


Figure 8-18. *Besides being a tacky user experience, we've opened ourselves to security risks by revealing a lot about the current process.*

We have a variety of options for handling errors. We could, for example, wrap the offending line in a *Try/Catch* block and perform some task when the exception is triggered. It's always good practice to wrap potentially sensitive code in *Try/Catch* blocks. However, we don't always have the luxury of knowing with such certainty when and how an error can be triggered. We can tell .NET that we have a custom error page we'd like to use when unhandled exceptions occur.

First, create a new file in the project called *Error.aspx*; add to the page the markup shown in Listing 8-26.

Listing 8-26. *The Error.aspx page Markup*

```
<%@ Page Language="IronPython" CodeFile="Error.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
    <title>An error has occurred.</title>
</head>
<body>
    <h1>
        Oops!</h1>
    <h2>
        Something totally unexpected happened; don't worry, we don't think it was➥
your fault.</h2>
    <p>
        An error happened that the application couldn't resolve automatically.➥
Please hit the Back button on your browser to try again, or➥
contact technical support if the problem persists. We're sorry for the➥
inconvenience.</p>
</body>
</html>
```

■ **Tip** It's always good to encourage your users at each step. If an error occurs, it doesn't hurt to reassure them that they didn't ruin everything or break the Internet. (You'd be surprised what people think they can do on a computer.) Always provide them with an action to take, preferably one that is simple and easy, and give them some way to communicate with an actual human being (if at all possible) to resolve the problem they're having. A little courtesy and reassurance goes a long way toward encouraging customer loyalty and repeat users, even in the face of errors. In an ideal world you'll be learning about the problem immediately via an automated e-mail that informs you of the problem when it happens; never underestimate the importance of an error log.

Now we need to create a bare-bones *Error.aspx.py* file that contains the code shown in Listing 8-27.

Listing 8-27. *The Beginning of Error.aspx.py*

```
def Page_Load(sender, e):
    pass
```

With a basic error page in hand, it's time to look into a project file we've overlooked so far: the almighty *web.config* file. This file contains a wide variety of configuration settings for your application. It is in an XML format (everything seems to come back to XML, does it not?), and, honestly, you won't modify many of its sections too often. Some of it is really boilerplate that doesn't need modification; by modifying it, your program wouldn't

run! Don't worry though, learning your way around isn't too difficult, and setting up error handling is simple.

Open the *web.config* file. You should see quite a bit of configuration information. Scroll down until you find the block of text shown in Listing 8-28.

Listing 8-28. *The <customErrors> Tag in web.config*

```
<!--
            The <customErrors> section enables configuration
            of what to do if/when an unhandled error occurs
            during the execution of a request. Specifically,
            it enables you to configure HTML error pages
            to be displayed in place of an error stack trace.

        <customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
            <error statusCode="403" redirect="NoAccess.htm" />
            <error statusCode="404" redirect="FileNotFound.htm" />
        </customErrors>
 -->
```

Visual Studio has kindly added a few comments to explain what the *customErrors* tag does, but they've also commented out the *customErrors* section entirely! Let's adjust this section to look like Listing 8-29.

Listing 8-29. *The <customErrors> Tag in web.config, Configured to Use Our Page*

```
<!--
            The <customErrors> section enables configuration
            of what to do if/when an unhandled error occurs
            during the execution of a request. Specifically,
            it enables you to configure HTML error pages
            to be displayed in place of an error stack trace.
-->

        <customErrors mode="On" defaultRedirect="Error.aspx">
       </customErrors>
```

■ **Note** What happened to the other error codes, like 403 and 404? There are a variety of error codes in the HTTP standard; 404, for example, indicates that a file could not be found. Right now, we don't have pages to support a variety of error codes, so it is better just to send all errors and exceptions to the *Error.aspx* page. You can go back at any time and add page-specific error handling to this section.

That's it! You can run the application again and submit some information. The IDE will likely interrupt you to inform you of the exception, but close that window and continue execution of the program by pressing F5. You should be presented with the custom error page we created before (Figure 8-19). This is a friendlier user experience. Also, we have not exposed any specific details about the underlying code or implementation, which leaves us in a better security position.



Figure 8-19. *Now the user doesn't get slapped in the face with technical mumbo-jumbo.*

■ **Tip** The difference between `mode="On"` and `mode="RemoteOnly"` in the *customErrors* tag is that if the mode is set to *RemoteOnly*, then logging onto the server where the code is hosted and browsing the page will show you the ugly yellow error page, but users browsing your site from their own machines will see the friendly error page. Setting it to *On* means that everyone will see the friendly error page, regardless of where they're located. Normally you'll want customErrors set to *RemoteOnly* in production so that you can conveniently hop onto the server and reproduce the error message yourself.

Subtle Security Flaws

One of the most important things to remember about security and error handling is that just because things seem to be operating properly doesn't necessarily indicate that things are perfect. Security is often an ongoing task, and certainly neither you nor your users

benefit from the mentality that you've got things locked down. It's better to assume there is a tiny flaw somewhere and to continue trying to improve the situation when things are calm than when that flaw has already been exposed.

In my experience, one of the most difficult security flaws to identify is the cross-site scripting (XSS) attack. This type of attack can take a variety of forms, some of them simple and some extremely complicated (and, dare I say, ingenious). The absolute simplest attack is that of *session stealing*. In a session stealing attack:

1. User A logs onto a site with his or her credentials.
2. User A browses a page that contains some form of malicious code created by User B.
3. When User A browses the page with malicious code, it is executed and User A's security information is sent to User B without User A's knowledge.
4. User B is now able to operate on the site as User A.

If User A reaches step 4, then he has little way of defending himself and stopping User B from wreaking havoc with his personal information. This attack is more common than you'd think, in part because it is easy to overlook these types of security issues, particularly as web applications become increasingly complex. How easy is it, really?

■**Note** You could argue that we're manufacturing these situations ourselves and that if we hadn't gone out of our way to create these problems, they wouldn't exist. Up to a point, that is true; that is, we are creating security flaws for demonstration purposes. It is very simple to make some of these mistakes while working on your applications. I would prefer to create them artificially in a safe environment and show you ways to mitigate these problems *before* you've launched code to production and have your users (and most likely your boss) up in arms when things go sour.

Arbitrary Code Execution

We can run a quick example that proves how easy this sort of attack is. Let's open *thankYou.aspx.py* again and modify it so that it looks like Listing 8-30.

Listing 8-30. *Adding Some Insecure Code to Be Displayed to the User in thankYou.aspx.py*

```
def Page_Load(sender, e):
    prevTxtName = PreviousPage.FindControl("txtName")
    prevTxtPassword = PreviousPage.FindControl("txtPassword")
    litFeedback.Text = prevTxtName.Text + prevTxtPassword.Text
    Response.Write("<script type=\"text/javascript\">var _0x3774=➥
[\"\x48\x65\x6C\x6C➥
\x6F\x20\x69\x6E\x73\x65\x63\x75\x72\x65\x20\x77\x6F\x72\x6C\x64\x21\"];➥
alert(_0x3774[0x0]);</script>");
    pass
```

If you run the application and submit information to the *Default* page, then after you click the Submit button you should see what is shown in Figure 8-20.

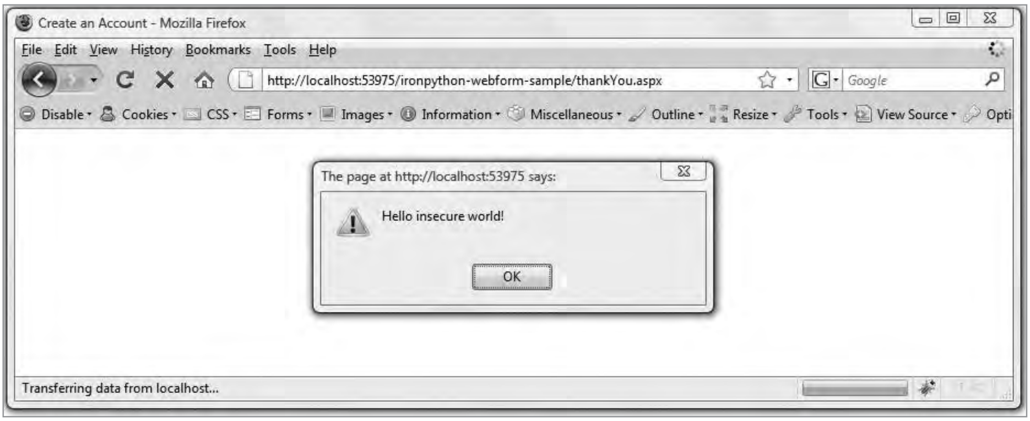


Figure 8-20. *We executed some unsafe code!*

■ **Note** You can use *Response.Write* to write directly to the output stream. If you were to *Response.Write* some plain text instead of the JavaScript we supplied, you'd see it immediately at the top of the page, before all other content. I find it's frequently a useful method for debugging; in particular it's convenient for displaying values to the page for testing purposes.

If a user had gotten that particular string onto one of your pages or in a database, for example, it would get executed every time a user browsed that page. In our example, we definitely alerted the user to the existence of the code: a big alert box shows up to notify

the user that something has happened (we're even rubbing it in!). Normally, things aren't so obvious; in fact, most attacks are designed so that things appear to be operating normally and the user is none the wiser that anything is up. We developers are not powerless in the fight. Truth be told, taking the wind out of the attack's sails is pretty easy.

Open *thankYou.aspx.py* again and make the modification shown in Listing 8-31. Then run the application again (Figure 8-21).

Listing 8-31. *Adding Some Insecure Code to Be Displayed to the User*

```
def Page_Load(sender, e):
    prevTxtName = PreviousPage.FindControl("txtName")
    prevTxtPassword = PreviousPage.FindControl("txtPassword")
    litFeedback.Text = prevTxtName.Text + prevTxtPassword.Text
    Response.Write(Server.HtmlEncode("<script type=➥
\"text/javascript\">var _0x3774=[➥
\"\x48\x65\x6C\x6C\x6F\x20\x69\x6E\x73\x65\x63\x75\x72\x65\x20\x77\x6F\x72\x6C➥
\x64\x21\"];alert(_0x3774[0x0]);</script>"));
    pass
```

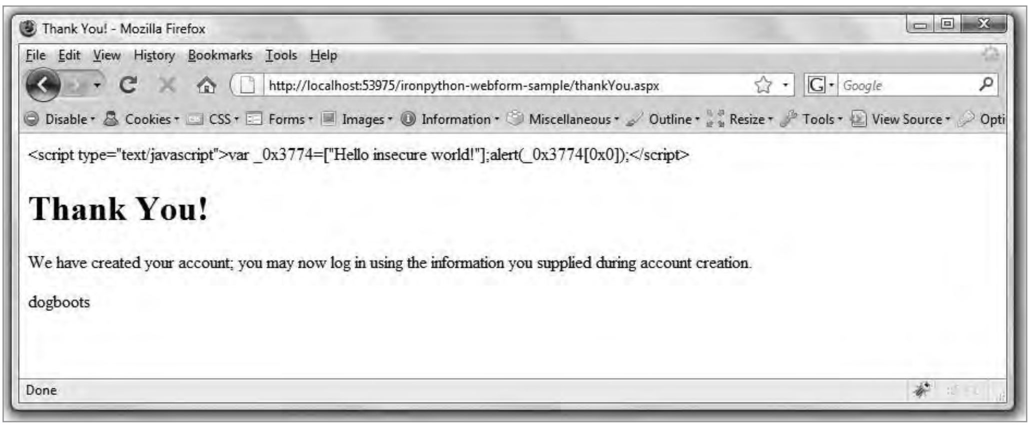


Figure 8-21. *Server.HtmlEncode can offer a pretty good degree of protection.*

Now, not only does the alert box not pop up, but we're actually presented with the text of the *Response.Write* call on the page itself. We can also see that the *Server.HtmlEncode* method has taken the previously obfuscated values and substituted the end result "Hello insecure world!" A quick look at the source code (Listing 8-32) reveals what's happened.

■Tip As with most operations, there is a cleverly named method that does the exact opposite. *Server.Html-Decode* will take an HTML-encoded string and revert it to executable markup. In some instances you'd want this behavior, but in not too many. It's just nice to know it exists.

Listing 8-32. *Many of the Characters Have Been HTML Encoded, Preventing Them from Being Executed as Client-Side Code.*

```
&lt;script type="text/javascript"&gt;var _0x3774=["Hello insecure➥
world!"];alert(_0x3774[0x0]);&lt;/script&gt;
```

We established in Chapter 7 that an extremely good rule of thumb is to assume that all user input is unsafe. Hopefully what we just did reinforces that; our code was a nuisance, displaying an obnoxious message to the user. Malicious code could (and frequently does) have disastrous results.

Summary

Developing web pages in IronPython is not only easy, but powerful. Although developing applications for the web requires a different approach and set of considerations, IronPython benefits from the underlying .NET framework and the consistency therein, meaning that web development is a distant cousin to desktop development, and your IronPython experience transfers readily between the two. We created a simple web application, learned about ViewState and how to use it to store and retrieve data, showed the limitations of the PostBack model and how to overcome them, learned about data validation and custom error pages, and finally covered how to protect ourselves against displaying unsafe content to our users.

CHAPTER 9

IronPython Recipes

"Without requirements or design, programming is the art of adding bugs to an empty text file."

— Louis Srygley

Many times in your development career, you'll find yourself in a position where it seems like someone just *had* to have solved a particular problem; in a lot of cases you'd be right. There's no need to reinvent the wheel when you could simply make use of it. This chapter is all about those recipes for code reuse as well as useful code nuggets in general. Hopefully, as you become more experienced as an IronPython developer, you'll find some useful at-a-glance tips that make your life a bit easier.

How to Use This Chapter

This chapter is all about small, easy-to-digest bits of IronPython code. The idea is that you should be able to take any of these snippets and use them quickly, with little to no up-front work on your end. You can use many of the snippets in multiple places; for instance, converting between data types is done identically whether you're working on a console application or a web application. There are specific bits of code for web and desktop applications that won't translate from one to another, but I've tried to keep things distinct and provide helpful road markers so that you can get the information you want quickly. With that in mind, let's jump in the deep end!

Note I should also mention that occasionally you'll see some repeated code; for example, adding the CLR and System namespaces to applications. These required boilerplate tasks need to be performed in many cases, and I would prefer that the code snippets are "fire and forget" so that you don't have to worry about what namespaces need to be imported where. So forgive a little repetition, but it should make it easier to flip to any page and make use of the code examples. If I don't include a line you might have seen elsewhere, you don't need it to use the snippet; in short, batteries included!

Until otherwise noted, the following examples can all be run directly from the IronPython interpreter. If you need a reminder, you can find it under the IronPython installation directory (for me this is C:\Program Files\IronPython 2.0); the interpreter application is called *ipy.eye*. Some examples may include the words *(press Enter)*, placed at points where the interpreter is waiting for your continued input, such as when defining classes and methods. I included it so that you and the interpreter wouldn't have an unnecessary staring contest. I've found that, despite the fact I have four eyes, the interpreter continues to win that particular fight.

Displaying the String Representation of an Object

Frequently, it is useful to display the string representation of an object (Listing 9-1). Earlier in the book we looked at adding custom methods to our classes so that we could easily call one method and get a string value back regarding the state of the value. In .NET, objects by default inherit the ability to call the *.ToString()* method and get a string representation of that object back. Since *object* is the base class for all .NET data types, that functionality is exposed across those more specific types.

Listing 9-1. *Displaying the String Representation of an Object*

```
>>> import clr
>>> clr.AddReference("System")
>>> myAge = 25
>>> print myAge.ToString()
25
>>> class Human:
...     age = 25
...     name = "Alan Harris"
... (press Enter)
>>> me = Human()
>>> print me
<__main__.Human instance at 0x000000000000002B>
>>> print me.ToString()
<__main__.Human instance at 0x000000000000002B>
```

Tip Earlier in the book I mentioned how useful it is to provide your own *.ToString()* method for convenience. This is the proof; using *.ToString()* on a class returns the instance of the class but little other useful information. See the discussion later in this chapter on "Implementing Your Own *.ToString()* Method."

Converting Between Two Base Data Types

Besides displaying the string representation of a value, sometimes it may be necessary to convert a value of one type to another (Listing 9-2). The *Convert* class in the *System* namespace provides a variety of conversion types that are easily accessible. There are a few caveats, however.

- A conversion method exists for each base data type: Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime, and String.
- Every base type can be converted to every other base type, but these conversions can throw exceptions. The act of converting does not mean the conversion is going to be successful.
- Conversions will fail if the data types are incompatible, such as converting a *Char* to a *DateTime* object.
- Conversions will fail if the final data type results in a loss of data. For example, the maximum value for an Int32 is larger than the maximum value for a Byte. Converting from an Int32 with a value of 2,147,483,647 to a Byte will fail, because the Byte's maximum value is 255.

Listing 9-2. *Converting Between Two Base Data Types*

```
>>> import clr
>>> clr.AddReference("System")
>>> import System
>>> myFloat = 1.23456789
>>> print System.Convert.ToInt32(myFloat)
1
>>> print System.Convert.ToBoolean(myFloat)
True
>>> print System.Convert.ToString(myFloat)
1.23456789
>>> myFloat2 = 0.0
>>> print System.Convert.ToBoolean(myFloat2)
False
```

Note Two things worth noting from this example are that calling *System.Convert.ToString()* on a data type is functionally equivalent to calling the *.ToString()* method on the data type itself. The shorthand version is simply more convenient in most cases. Also note that *.ToBoolean()* will return False *only* when the value of the data type is precisely 0. Nonzero values, even negative ones, will return True. I don't expect you'll be making that sort of conversion often, but the behavior can be surprising if you're not aware of it beforehand.

Implementing Your Own *.ToString()* Method

As we saw in an earlier example, the default implementation of *.ToString()* is not terribly useful when applied to classes we've created. As a convenience to yourself and other developers, it's generally a good idea to provide your own *.ToString()* method with your classes (Listing 9-3). It makes debugging easier; you can simply call *.ToString()* on an instance of a class and get back any relevant data in one line, and it conforms to a coding standard that other .NET developers are used to.

Listing 9-3. *Implementing Your Own .ToString() Method*

```
>>> import clr
>>> clr.AddReference("System")
>>> import System
>>> class Foo:
...     age = 30
...     name = "Joe Plumber"
...     def ToString(self):
...             return "age: " + age.ToString() + " - name: " + name
... (press Enter)
>>> bar = Foo()
>>> print bar.ToString()
age: 30 - name: Joe Plumber
```

Note that when we called *.ToString()* on the *age* variable, the default implementation for an integer was used. This is why providing your own in a custom class is almost never a bad idea; it will only apply to the instance of the class itself, leaving all other classes and variables alone. This is a simple method to implement on most classes, and it will pay for itself quickly.

Inheriting from a Base Class

By design, one of the intrinsic strengths of object-oriented design is the ability for one class to inherit from another and to take on the attributes of that class through "is a" and "has a" relationships (e.g., a car "is a" vehicle and "has" wheels). Establishing a base class for other classes to inherit from is exceedingly simple in IronPython (Listing 9-4).

Listing 9-4. *Inheriting from a Base Class*

```
>>> class Vehicle:
...     weight = 100
...     def SetWeight(self, value):
...             self.weight = value
... (press Enter)
>>> class Car(Vehicle):
...     wheels = 4
...     def SetWheels(self, value):
...             self.wheels = value
... (press Enter)
>>> auto = Car()
>>> print auto.weight
100
>>> print auto.wheels
4
>>> auto.SetWeight(150)
>>> auto.SetWheels(6)
>>> print auto.weight
150
>>> print auto.wheels
6
```

The *Car* class inherits from the *Vehicle* class and gets access to both the properties and methods that the *Vehicle* class provides. We didn't have to implement either the *weight* property or the *SetWeight* method; they were pulled in automatically. Used properly, this can be an incredibly effective software design technique because an effective design allows you to use parent and child classes interchangeably in many cases, which proves to be a terrific foundation for component-based development.

Getting User Input from the Console

Retrieving user input from the console is a frequent task of console applications. The safe way to perform this task is via the *raw_input()* method (Listing 9-5). Note that in displaying the results back to the user, we will also call the *.ToString()* method as demonstrated in Listing 9-1 so that the interpreter understands that we want to treat the *age* variable as a string and to concatenate it, not add it to some other value.

Listing 9-5. *Getting User Input from the Console*

```
>>> firstName = raw_input("What is your first name? ")
What is your first name? Alan
>>> lastName = raw_input("What is your last name? ")
What is your last name? Harris
>>> age = raw_input("What is your age? ")
What is your age? 25
>>> print firstName + " " + lastName + " is " + age.ToString() + " years old."
Alan Harris is 25 years old.
```

Concatenating Strings Efficiently with the *StringBuilder*

In general, working with strings is a pretty slow set of operations. It can also be fairly costly, because strings in .NET are immutable. When you concatenate strings, what's really happening is that multiple strings are created in the background, and the final result is a brand new string. This can be very wasteful, particularly if it's performed in some type of loop. A better method that .NET provides is the *StringBuilder* class, which you can use to concatenate strings together without creating multiple strings that exist only to be deleted (Listing 9-6). Generally the last step in applying the StringBuilder is converting the final value to a string to be used elsewhere in your code.

Listing 9-6. *Concatenating Strings Efficiently with the StringBuilder*

```
>>> import clr
>>> clr.AddReference("System")
>>> import System
>>> import System.Text
>>> sb = System.Text.StringBuilder()
>>> sb.Append("This")
```

```
<System.Text.StringBuilder object at 0x000000000000002B [This]>
>>> sb.Append(" ")
<System.Text.StringBuilder object at 0x000000000000002B [This ]>
>>> sb.Append("appends")
<System.Text.StringBuilder object at 0x000000000000002B [This appends]>
>>> sb.Append(" ")
<System.Text.StringBuilder object at 0x000000000000002B [This appends ]>
>>> sb.Append("without additional strings.")
<System.Text.StringBuilder object at 0x000000000000002B➥
[This appends without additional strings.]>
>>> print sb.ToString()
This appends without additional strings.
```

■ **Tip** This isn't going to make or break your application performance in 99.99% of your coding life. If string concatenation is bringing your application's performance down, it's likely that the entire design needs to be examined. However, using the *StringBuilder* is not a bad practice to get into; it doesn't cost you anything other than a bit more typing, and it keeps your underlying memory usage cleaner. Concatenating strings together with the + symbol *always* allocates new memory, but the *StringBuilder* will allocate new memory only if the existing object buffer hasn't enough space left for what you're adding. Note that the address of the object is always the same for the *StringBuilder* (in my case, the location was 2B).

I have heard it argued that the CLR will frequently make a more intelligent decision and occasionally, behind the scenes, convert string concatenation code to a *StringBuilder* style of concatenation for you. Though it's not a great idea to think you're smarter than the compiler, it is generally a good practice to try and help it along where possible with smart design decisions. A good rule of thumb is that if you know precisely how many strings you're concatenating, it's probably safe to use the regular old string way, because the CLR will often translate that into a more efficient structure automatically. If you are combining some arbitrary number, stick with the *StringBuilder*. For more information on the topic, visit the MSDN documentation for the *StringBuilder* class, which is currently located at `http://msdn.microsoft.com/en-us/library/system.text.stringbuilder.aspx`.

Creating a Set of Enumerations

One thing that modern developers shy away from when possible is the concept of *magic numbers*. Magic numbers are values that appear in source code as hard-coded constants. The numerical value by itself is meaningless. The enumerated value allows you to use a variable name instead of the specific value itself (Listing 9-7). In addition to providing clarity in your code, it makes maintenance easier if the value needs to change at a future date; you will have to change only the value of the enumeration in the single location in which it was defined, not throughout the entire application.

Listing 9-7. *Creating a Set of Enumerations*

```
>>> class Enumerations:
...     foo = 0
...     bar = 1
...     xyzzy = 2
... (press Enter)
>>> enumValue = Enumerations.foo
>>> print enumValue
0
>>> enumValue = Enumerations.xyzzy
>>> print enumValue
2
```

■ **Note** Random trivia: If you ever played the old text-based game *Colossal Cave Adventure*, you may recall *xyzzy* as the magic word that allows the player to teleport between two specific locations. Otherwise it produces the dry response "nothing happens." It's been a long-running joke among developers. Case in point: The *Minesweeper* game bundled with Windows has used it as a cheat code for quite some time.

Retrieving Command-Line Arguments

Many (I would even venture to say most) console applications accept one or more arguments passed via the command line. The IronPython interpreter is a perfect example; you can run *ipy.exe* by itself to start the interpreter, or you can run a specific script, as we will do in this example. You can access command-line arguments via the *System.Environment.GetCommandLineArgs()* method. Create a file called *CommandLineTest.py* in the *C:\Python* folder and add to it the code in Listing 9-8.

Listing 9-8. *Retrieving Command-Line Arguments*

```
import clr
clr.AddReference("System")
import System
```

```
class CommandLineTest:
      cmds = System.Environment.GetCommandLineArgs()
      print cmds
      print cmds.Length
      if cmds.Length < 3:
            print "You did not specify an argument for this program."
      else:
            print cmds[2]

CommandLineTest()
```

The code will get a list of command-line arguments, display those arguments back to the user, display the number of arguments in the list, and display different output, depending on whether there are fewer than three arguments provided. Why three? Let's run the example two different ways to examine the output.

```
C:\Program Files\IronPython 2.0>ipy c:\python\CommandLineTest.py 1234
Array[str](('ipy', 'c:\\python\\CommandLineTest.py', '1234'))
3
1234

C:\Program Files\IronPython 2.0>ipy c:\python\CommandLineTest.py
Array[str](('ipy', 'c:\\python\\CommandLineTest.py'))
2
You did not specify an argument for this program.
```

Aha! The *GetCommandLineArgs()* method includes in the array of arguments the name of the executable, the script being called, *and* your parameters. Therefore, when checking command-line arguments in this fashion, start your counting at array element 2 (bearing in mind that arrays in IronPython, as in C, begin at 0 instead of 1). Depending on whether you're working with console, desktop, or web applications, you may find you have a greater or lesser number of elements in the command line.

Listing All the Files in a Folder

There are many circumstances in which you might want to list all the files in a particular folder. One that immediately springs to mind is one in which I found myself not long ago: I used IronPython as a scripting language for an existing application as a sort of plug-in

system (see Chapter 6 for some implementation ideas and examples), and I wanted to list all the custom IronPython plug-ins in a particular set of folders. Listing 9-9 shows how easy it is to access this information in IronPython.

Listing 9-9. *List All the Files in a Folder*

```
>>> import clr
>>> clr.AddReference("System")
>>> import System
>>> import System.IO
>>> di = System.IO.DirectoryInfo("c:\python")
>>> files = di.GetFiles("*.*")
>>> for file in files:
...     print file.Name
... (press Enter)
CommandLineTest.py
form.py
helloConditional.py
helloDynamic.py
helloWorld.py
humanBeing.py
humanBeingTest.py
static.py
test - Copy.p2
test.py
```

The *.GetFiles()* method accepts a string for a parameter; this parameter represents the filter you wish to apply to the folder. You can search for all files, files with a specific extension, files with a certain character in the title, and so on.

Conveniently Check the State of a String

A lot of data validation effort goes into checking whether strings (1) have been initialized to any value (meaning a non-null value) and (2) have an actual value. Note that there is a difference between a string that is null and a string with a value of “". Although they seem similar, the string set to null has not been initialized with any value whatsoever; the string with a value of “" is simply blank (aka empty).

.NET provides a convenient method called *String.IsNullOrEmpty()* that performs both checks for you, saving you a bit of typing and allowing you to perform this type of check consistently across all your application code (Listing 9-10).

Listing 9-10. *Conveniently Check the State of a String*

```
>>> import clr
>>> clr.AddReference("System")
>>> import System
>>> myString = ""
>>> if (System.String.IsNullOrEmpty(myString)):
...     print "myString is null or empty."
... else:
...     print myString
... (press Enter)
myString is null or empty.
```

■ **Tip** There used to be a pretty nasty bug in .NET where under certain conditions the *.IsNullOrEmpty()* method would actually trigger a *NullReferenceException*. If you think about that for a few seconds, you'll realize how annoying it would be to get a *NullReferenceException* when you're performing a check for null values that is provided by .NET itself. Luckily, that bug was corrected as of .NET 2.0 Service Pack 1. If you're using any version of .NET that is 2.0 SP1 or higher, you shouldn't encounter this issue. For more information on the bug and its fix, take a look at Microsoft Help and Support's article on the topic, which is currently located at `http://support.microsoft.com/kb/940900/`.

If you are (at some point) using a version of .NET that is 2.0 or earlier, you probably won't encounter this one, unless you're running *.IsNullOrEmpty()* inside of a loop. It's good to know anyway, in case you ever do come across it.

Implementing the Singleton Design Pattern

Sometimes in your applications you will have objects that by their very nature should exist at most one time and one time only; a global application object would be one example. (Indeed, Singletons occasionally get a bad rap for being glorified global variables, which developers frown on because it's more of a hassle to ensure that the state of that variable is correct everywhere.)

In IronPython, we'll store an instance of a class in a variable for reference; if that instance variable is null, we know we haven't yet created an instance of the singleton. If it's not null, we should reuse the existing reference instead of creating a new one.

Create a new file in your *C:\Python* folder called *singleton.py,* and enter the code in Listing 9-11.

Listing 9-11. *Implement the Singleton Design Pattern*

```
class SpecialResource:
    class Singleton:
        def identifyMyself(self):
            return id(self)

    __singletonInstance = None

    def __init__(self):
        if SpecialResource.__singletonInstance is None:
            SpecialResource.__singletonInstance = SpecialResource.Singleton()

    def __getattr__(self, attr):
       return getattr(self.__singletonInstance, attr)

resourceOne = SpecialResource()
print resourceOne.identifyMyself()

resourceTwo = SpecialResource()
print resourceTwo.identifyMyself()

resourceThree = SpecialResource()
print resourceThree.identifyMyself()
```

```
c:\Program Files\IronPython 2.0>ipy c:\python\singleton.py
43
43
43
```

You can see that despite having created three different objects, the *Singleton* class permits only one true instance to be created; as a result, the output of all three objects is identical. The specific number you see may differ from mine. The important point is that the numbers match one another in your output. If they were different, it would indicate that there was a unique ID for each object, and therefore the objects would not be Singletons. Also remember that the value displayed is not a value contained by an object itself; rather, it is the ID for the object itself.

■ **Tip** As you become more advanced as an IronPython developer, you may find your attention turning to multithreaded software development (particularly since multicore processors are becoming the norm, even on lower-end machines); the code for the Singleton implementation is not "thread safe." Although a proper discussion of multithreaded development is a topic worthy of a (very large) book by itself, it is sufficient to say that if you apply this code across multiple threads, you may encounter situations where two threads are creating instances of the Singleton at almost the same time. This condition can actually violate the design rules of the Singleton pattern and allow *two* different instances to be created. You can achieve a simple solution by locking code that modifies the private *__singletonInstance* variable, thereby restricting access to the current thread alone; the second thread to attempt to create an instance will see the correct result. These types of threading conditions can be very difficult to diagnose because by definition they are random and not easily reproducible.

Opening a Connection to a Database

Databases such as SQL Server and Oracle serve as repositories of data; as such, they are exceedingly effective at both fast and powerful data access. Communicating with these databases and performing data-related tasks is initiated via database connection objects (Listing 9-12). .NET already contains a variety of data providers for various database systems; it also offers generic data providers if you're communicating with a system that does not already have built-in .NET support. In general, you should use the data provider specific to the system with which you're communicating because the various tools in .NET are configured to be as quick as possible for those particular systems. Thus, you should absolutely take advantage of that performance effort.

Throughout the book we've been using SQL Server, in part because it's a very solid database system and also because Microsoft has free versions available for developers, which creates a low barrier to entry.

Listing 9-12. *Open a Connection to a Database*

```
>>> import clr
>>> clr.AddReference("System.Data")
>>> from System.Data import *
>>> from System.Data.SqlClient import *
>>> conn = SqlConnection("Data Source=ALAN-DEVPC\SQLEXPRESS;➥
Integrated Security=True;Initial Catalog=IronPython;User Instance=false")
>>> comm = SqlCommand("SELECT name FROM Employee", conn)
>>> conn.Open()
```

```
>>> reader = comm.ExecuteReader()
>>> if (reader.Read()):
…     print (reader.GetString(0))
… (press Enter here)
Alan Harris
>>> reader.Close()
>>> conn.Close()
```

■Note Listing 9-12 is based on a table and data set that we created in Chapter 7. If you have not created this table or populated it with any data, then flip back to that chapter and do so. It only takes a few minutes and it's nice to have some dummy data with which to work.

Performing a Bubble Sort on a Set of Elements

There are many, many ways to sort a set of elements, with varied results in terms of performance. In computer science courses, the bubble sort is frequently introduced first among sort algorithms because it is an easy sort to explain and implement. Unfortunately, as the number of elements in your input set increases, performance decreases significantly. Also, the initial sorting of the elements plays a role; if the elements are positioned very far from where they will be when in their final order, then performance will be slower because more computations have to be performed.

Even so, a bubble sort is still a valid sorting methodology and a good algorithm to explore. Essentially, you are looping through a set of elements, comparing the current element to the next one in the set. If the next one has a value less than the current one, you swap the two positions. You continue this process until the elements are in order from left to right (Listing 9-13).

Listing 9-13. *Perform a Bubble Sort on a Set of Elements*

```
>>> def bubbleSort(elementList):
...     # get the number of elements (REMEMBER: zero-based array,➥
so it is length - 1)
...     maxIndex = len(elementList) - 1
...     # a flag to set whether the sorting is complete
...     unsorted = False
...     while not unsorted:
```

```
...             unsorted = True
...             # for each element in the array, we need to do a comparison
...             for index in range(maxIndex):
...                     # if the next element is less than➥
or equal to the current one...
...                     if elementList[index + 1] <= elementList[index]:
...                             # ...reverse them and set the flag so that➥
the procedure continues.
...                             elementList[index], elementList[index + 1] =➥
elementList[index + 1], elementList[index]
...                             unsorted = False
...     return elementList
... (press Enter)
>>> myList = [1, 10, 394, 812, 9, 54]
>>> print bubbleSort(myList)
[1, 9, 10, 54, 394, 812]
```

Using the *StopWatch* Class to Time Operations

In terms of application performance, there are many considerations to account for. Performance might refer to the amount of resources consumed, the speed at which a particular operation completes, and so on. A simple way to evaluate, with reasonably high precision, the duration of time that an operation took is with the *StopWatch* class.

The *StopWatch* class exposes some fairly intuitive methods: Start and Stop, which are used for controlling operation of the StopWatch. The time is measured using a value of type *System.TimeSpan*, which lets you see precisely how long something took to complete.

Create a new file in your *C:\Python* folder and call it *stopWatchTest.py*. Enter the code shown in Listing 9-14.

Listing 9-14. *Use the StopWatch Class to Time Operations*

```
import clr
clr.AddReference("System")
import System
import System.Diagnostics
```

```
def stopWatchTest():
      stopWatch = System.Diagnostics.Stopwatch()
      stopWatch.Start()
      print "This takes place during the timing period."
      stopWatch.Stop()
      print stopWatch.ElapsedMilliseconds.ToString() + " milliseconds to➥
perform the previous operation."
      print stopWatch.Elapsed.ToString() + " total time elapsed."

stopWatchTest()
```

```
c:\Program Files\IronPython 2.0>ipy c:\python\stopWatch.py
This takes place during the timing period.
12 milliseconds to perform the previous operation.
00:00:00.0121341 total time elapsed.
```

■**Note** The number of milliseconds it took for your operation to complete will likely be different than it took for mine. In fact, repeated runs will produce different results, depending on how busy your system is at the time of measurement. This variance is totally normal.

Baking Cookies

Cookies have gotten a bad rap for a long time on the Internet, for some reasons justified and some unjustified. Cookies themselves are perfectly useful and safe; it's really the manner in which they're used that some find troublesome. Essentially a *cookie* is a small text file that a web application can employ to store and retrieve a bit of data from the user's hard drive (Listing 9-15). Used properly, they can offer some very beneficial functionality to modern web applications; used improperly, they open security holes and put your users' data and systems as risk.

■**Tip** I've harped on this a few times, but, at the risk of sounding like a broken record, *do not* store sensitive information in cookies. Treat cookies like objects that you wouldn't mind exposing to the whole world. Do you want the whole world seeing your credit card number? No? Then don't store it in a cookie. This is a great rule of thumb.

Listing 9-15. *Baking Cookies*

```
import System

def Page_Load(sender, e):
    Response.Cookies["ironPythonCookie"]["dateTimeValue"] =➥

System.DateTime.Now.ToString()
```

A good practice with cookies is to set an expiration date; after the date you select, the cookie will automatically be dropped by the user's browser. The date of expiration obviously varies, depending on the needs of your application; for this example, we'll set the expiration date to be one year in the future (Listing 9-16).

Listing 9-16. *Baking Cookies with an Expiration Date (cont.)*

```
import System

def Page_Load(sender, e):
    Response.Cookies["ironPythonCookie"]["dateTimeValue"] =➥
System.DateTime.Now.ToString()
    Response.Cookies["ironPythonCookie"].Expires = System.DateTime.Now.AddDays(365)
```

Now we can take a look at the contents of the cookie and see if we were successful (Figure 9-1).

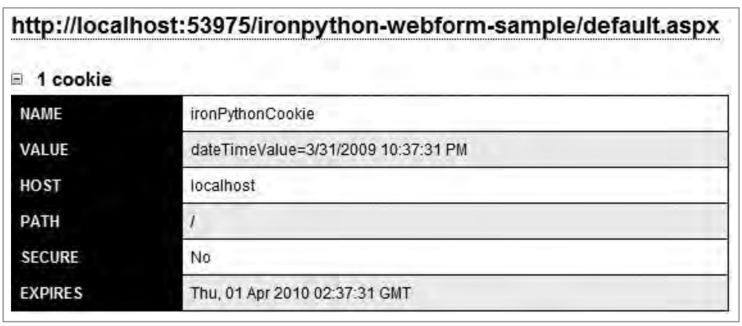


Figure 9-1. *Adding cookies to the current session was successful. Note the 2010 expiration date.*

■**Tip** How did I produce the output in Figure 9-1? I used the Web Developer Toolbar add-on for Firefox, available from `https://addons.mozilla.org/en-US/firefox/addon/60` (version 1.1.6) at the time of this writing. It's an extremely popular and powerful tool for use within Firefox that can save you a lot of development time; the best part is the price: totally free.

Reading Cookies

Storing information in a cookie is only half the battle (well, one-third if you consider deleting cookies, which we'll cover next). If you want to retrieve the value of a cookie, you should always check for whether that cookie exists before attempting to do so; if a user has disabled cookies, it won't exist and an exception will be thrown (Listing 9-17).

Listing 9-17. *Reading Cookies*

```
import System

def Page_Load(sender, e):
     Response.Cookies["ironPythonCookie"]["dateTimeValue"] =➥
System.DateTime.Now.ToString()

    # check for the existence of the cookie first
    if (Response.Cookies["ironPythonCookie"] == None):
        pass
    else:
        Response.Write(Response.Cookies["ironPythonCookie"]["dateTimeValue"])
```

■**Tip** Why not just wrap the call to read the cookie in a *Try-Catch-Finally* block? When we covered exceptions earlier in the book, I mentioned that it's generally not a good idea to structure your code around exceptions. A simple test for None (null) in this case works just fine. Exceptions, by definition, should exist for exceptional situations. A cookie that doesn't exist is not an exception situation; it's just a normal possibility that we should test for.

Deleting Cookies

It's all well and good to store cookies on a user's machine. But what happens if you want to delete that cookie for good?

Surprise—you can't! Actually, that's only half true. You cannot delete a cookie from the user's machine directly. What you *can* do is set the expiration date for that cookie to be in the past, which will trigger the user's browser to delete the cookie for you (Listing 9-18).

Listing 9-18. *Deleting Cookies*

```
import System

def Page_Load(sender, e):
    Response.Cookies["ironPythonCookie"].Expires = System.DateTime.Now.AddDays(-1)
```

Storing Data in Session State

HTTP, by definition, is a stateless protocol. This is why, for the longest time, web pages were static (and, frankly, pretty bland). We've come a long way, and server technology is such that we have options for persisting data between requests. In .NET, one of those ways is the *Session* state.

The *Session* state is a temporary repository in memory where you can store objects and data for use between multiple pages. It is best suited for small objects and data. Also, by default the session does have an expiration (usually about 20 minutes), so don't assume that information in the session is there.

Listing 9-19. *Store Data in Session State*

```
import System

def Page_Load(sender, e):
    Session["myName"] = "Alan Harris"
    Response.Write(Session["myName"])
```

Is the *Session* state the perfect solution in all cases? Not quite. Let's say you've created a three-page signup form for your snazzy new application, and you're storing all the data at each step in *Session* state. A user in the middle of filling out your awesome new signup form who walks away for 30 minutes will have to start from scratch if the session expires while the user is away.

Don't let this discourage you from using the *Session* state. It's more secure than cookies because the information contained within is never transmitted to the client (it's stored in server-side memory), and it can solve a good number of problems.

■ **Tip** Depending on your configuration and environment, you can actually persist session data to a database and essentially make it permanent, or you can store it in memory on a totally different server than the one your application is running on. Because of the additional configuration and software requirements, I've opted here to cover the simplest (and most common) session configuration. For more information on the ways *Session* can be configured, see the MSDN article on the topic at `http://msdn.microsoft.com/en-us/library/ms972429.aspx`.

Adding a Web Control Programmatically

In .NET, you can define pages dynamically at runtime; you don't necessarily have to construct the entire page in advance. The framework has a control called the *PlaceHolder* that serves as a container for the easy population of controls during runtime. The *PlaceHolder* control by itself has no appearance; when you add it to a page, precisely zero new markup will be sent to the user. It's purely an abstract container.

To use it, we first need to add it to the HTML markup of a page (Listing 9-20).

Listing 9-20. *Add a Web Control Programmatically Within Default.aspx*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Programmatic Controls</title>
</head>
<body>
    <form id="form1" runat="server">
         <asp:PlaceHolder ID="plc1" runat="server"></asp:PlaceHolder>
    </form>
</body>
</html>
```

In the code-behind file, we can now add controls to this *PlaceHolder* (Listing 9-21), and they will be rendered for the client.

Listing 9-21. *Add a Web Control Programmatically (cont.)*

```
import System

def Page_Load(sender, e):
    lblHello = System.Web.UI.WebControls.Label()
    lblHello.Text = "Hello World!"
    plc1.Controls.Add(lblHello)
```

When a user views the page, the *Label* control is rendered as a child of the *PlaceHolder* control with whatever properties applied that you set in code (Figure 9-2). You can add a variety of objects at runtime, ranging from simple to complex. This functionality (and the consequent ability to refer to these objects elsewhere in your code) permits extremely powerful page-creation options.

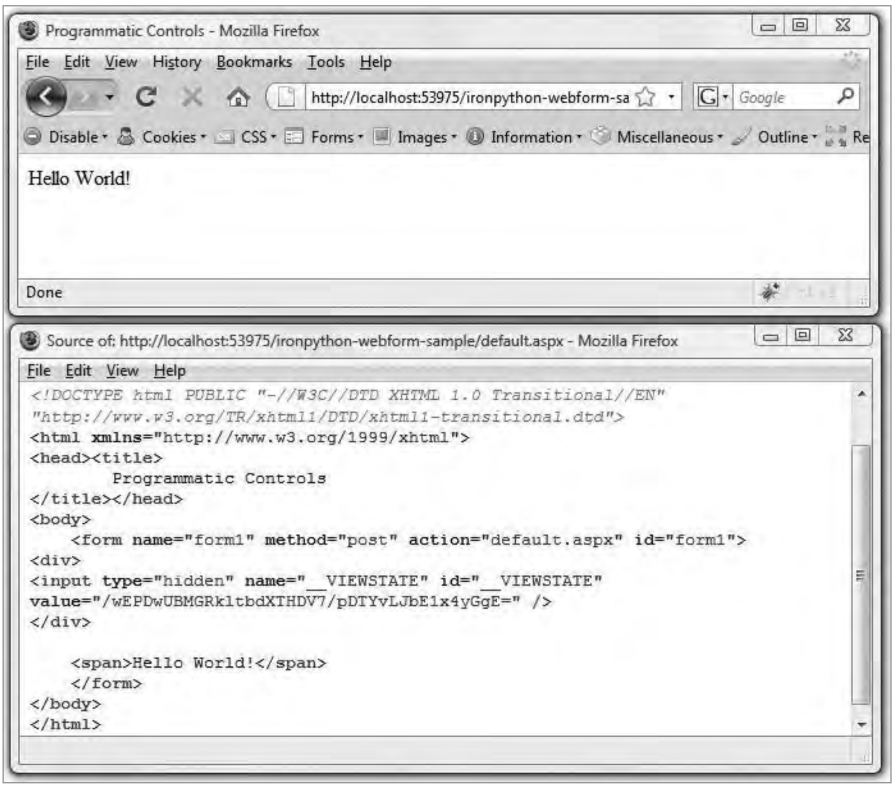


Figure 9-2. *The label is rendered, but notice that there's no markup for the PlaceHolder.*

■Note Did you notice the label rendered as a <span> tag? A *span* is an inline element with some specific behavior in terms of HTML and CSS control. You may not want spans in certain situations. The framework has a control called *HtmlGenericControl,* which we'll look at shortly, that permits more flexibility about exactly what gets output to the screen. Labels, however, will always render as spans.

Telling .NET to Render XHTML-Compliant Markup Using *Web.Config*

By default, .NET will send markup to the client that is somewhat "lowest common denominator." For various reasons you may want the markup to validate as XHTML Strict. Control over this is set in the *web.config* file in an element called *<xhtmlConformance>* (Listing 9-22). Place the tag anywhere inside the *<system.web>* element of your *web.config* file.

Listing 9-22. *Tell .NET to Render XHTML-Compliant Markup Using web.config*

```
<xhtmlConformance mode="Strict" />
```

■Tip You can also specify Transitional or Legacy, although my understanding is that there is some funniness with Legacy if you try to use ASP .NET AJAX. Check out Scott Guthrie's blog posting on the subject at `http://weblogs.asp.net/scottgu/archive/2006/12/10/gotcha-don-t-use-xhtmlconformance-mode-legacy-with-asp-net-ajax.aspx`.

How do we know that it worked? I added the tag to the *web.config* file for the project in the previous snippet, "Adding a Control Programmatically." If you look at the output markup for that snippet, you'll see that the *Form* tag has both a name and an *id* attribute; this is not valid for XHTML 1.0 Strict. Compare the results in Figure 9-3, where you will see that the *name* attribute is no longer in the *Form* tag. The output is now specified as being Strict compliant. Many times these changes are relatively minor, but sometimes they can be dramatic. If you plan to target XHTML 1.0 Strict, I recommend that you set that attribute in *web.config* as soon as you create your project. If you don't and you construct CSS to style your pages, you may find that CSS no longer works after applying that attribute, resulting in more work for you. Set it early and save yourself a headache (although you are by no means required to set it at all).



Figure 9-3. *The markup has been changed to be standards compliant.*

Custom HTML via the HtmlGenericControl

Earlier we looked at how to add a control to a web page programmatically; specifically we added a label, which was rendered between *span* tags. If you need a little more control over what markup is produced, you can use the *HtmlGenericControl* to output specific tags.

As before, we need to have a *PlaceHolder* on the page to contain the controls that we'll add (Listing 9-23).

Listing 9-23. *Custom HTML via the HtmlGenericControl*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Programmatic Controls</title>
</head>
<body>
    <form id="form1" runat="server">
         <asp:PlaceHolder ID="plc1" runat="server"></asp:PlaceHolder>
    </form>
</body>
</html>
```

With a *PlaceHolder* added, the syntax for adding an *HtmlGenericControl* is very similar to the method for adding any other control programmatically. Note that we can optionally specify the type of tag to render; in this case we'll render an h1 tag (Listing 9-24, Figure 9-4).

■ **Note** If you don't specify the type of tag to render, it will be a span by default.

Listing 9-24. *Custom HTML via the HtmlGenericControl (cont.)*

```
import System

def Page_Load(sender, e):
    lblHello = System.Web.UI.HtmlControls.HtmlGenericControl("h1")
    lblHello.InnerHtml = "Hello World!"
    plc1.Controls.Add(lblHello)
```

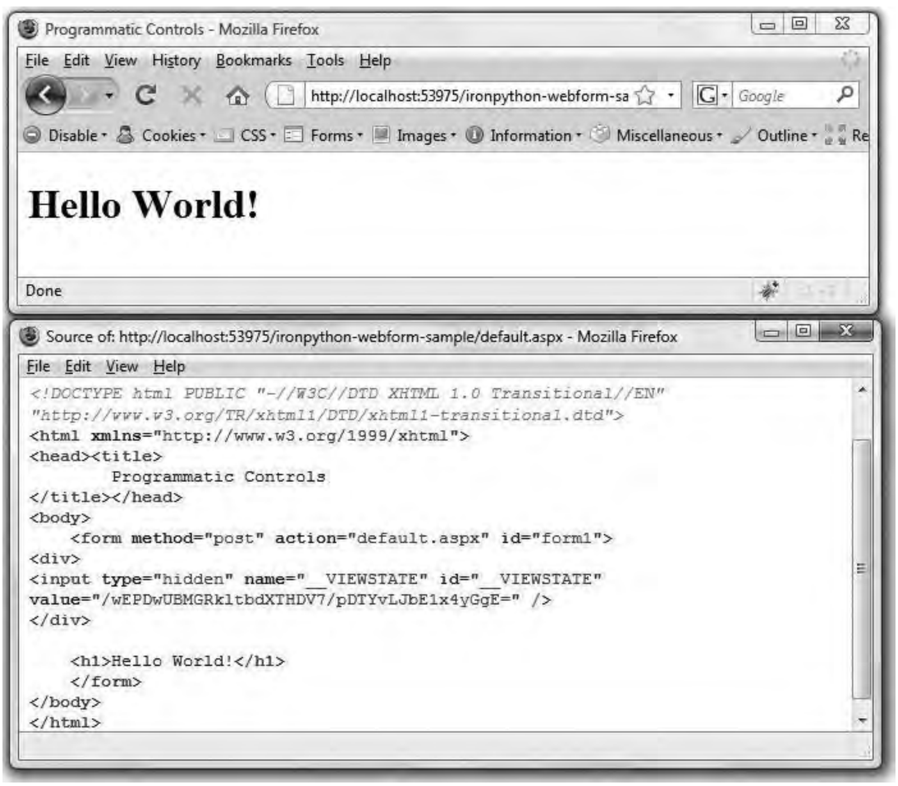


Figure 9-4. *The HtmlGenericControl renders as the type we specified as a parameter.*

Passing Information via the *QueryString*

The *QueryString* portion of a URL is the segment past the file extension, if one exists. In a URL such as `http://www.domain.com/default.aspx?id=1&dummy=2`, the *QueryString* parameters are *id* and *dummy*. We can pass information between pages using the *QueryString* (Listing 9-25, Figure 9-5).

It's always a good idea to check whether a *QueryString* parameter is null before attempting to perform operations on it. Also, since the *QueryString* is publicly visible both when a user hovers over a link and in the address bar once the user has browsed to that link, it is not at all suitable for sensitive information.

Listing 9-25. *Pass Information via the QueryString*

```
import System

def Page_Load(sender, e):
    if (Request.QueryString["id"] <> None):
         Response.Write("The QueryString value for the id parameter➥
is " + Request.QueryString["id"])
    else:
         pass
```

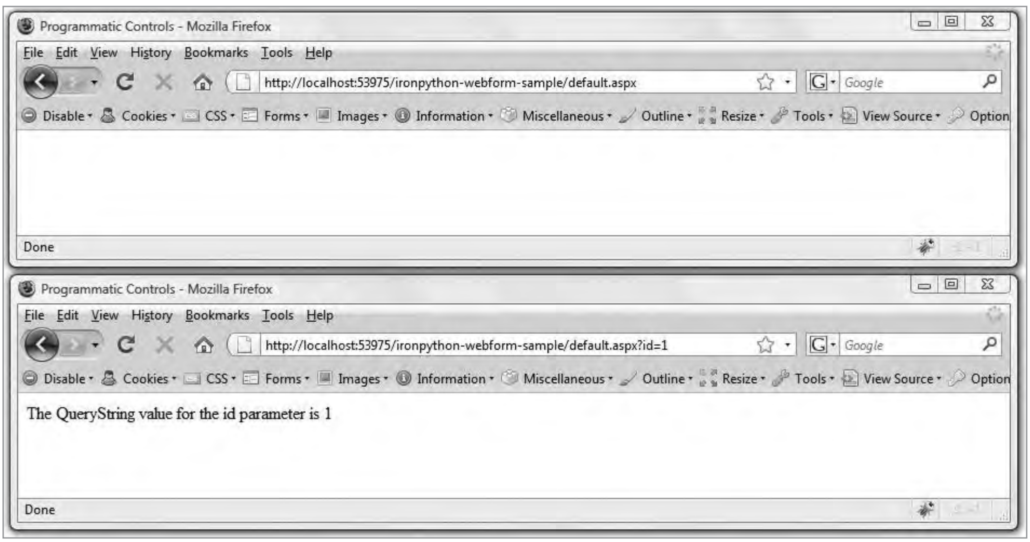


Figure 9-5. *You can use the QueryString to pass information between pages.*

Caching In

Arguably, web performance these days is bolstered in large part by intelligent caching. Data and pages that are *cached* are stored in memory or on a physical drive for faster retrieval so as to not be reconstructed on every request. You might consider storing a result set from a search in the cache so that users paging through the results don't constantly need to query the database on each page. It can greatly reduce the load on your servers, and in high-traffic situations *will* make or break your site. Educated (but judicious) use of caching can be a performance lifesaver. Luckily, in .NET it's exceedingly easy (Listing 9-26).

The cache, like Session state, has an expiration that you can adjust (but certainly don't disable it or set it to some extraordinarily long lifespan; that's a quick way to exhaust your resources and to leave you worse off than when you started).

Listing 9-26. *Caching In*

```
import System

def Page_Load(sender, e):
    if (Cache["id"] <> None):
        Response.Write("The cache currently has a value of "➥
+ Cache["id"].ToString() + " for the id variable.")
    else:
        Cache["id"] = 12345
        Response.Write("The cache has had a value inserted.")
```

■Tip Here's one of those times when checking for null values is really important. If you call *.ToString()* on the Cache["id"] element without any value present, .NET will throw an exception and you'll get the really ugly yellow screen of death.

The first time you run the code in Listing 9-26, nothing in the cache will be called *id*, so you will see the output displayed in Figure 9-6.

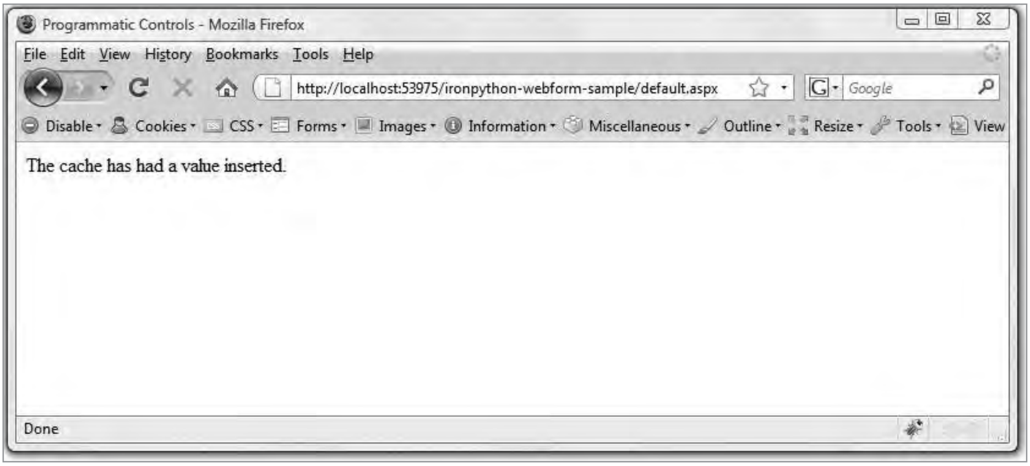


Figure 9-6. *Caching a value on the first run*

If you refresh your screen, you should see instead the output displayed in Figure 9-7, indicating that the code is retrieving the appropriate information from the cache.

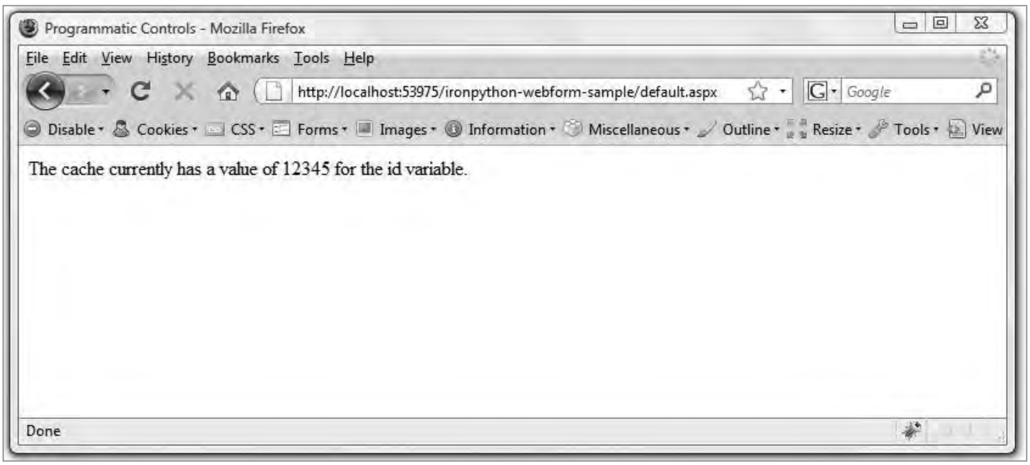


Figure 9-7. *Retrieving data from the cache*

■ **Tip** The .NET cache is stored in the running ASP .NET worker process. When you are running a web application that operates on one box only, the cache works fine. If you are scaling out to operate across many web servers, a distributed cache solution is a better bet. With the .NET cache, your cache hit ratio in that situation will likely be too low to even consider using.

Setting HTML Attributes at Runtime

In certain situations, as a developer you'd like to perform tasks at runtime on a specific control; perhaps you like to attach a little JavaScript to a button or to change the CSS class of a text box, and so on. .NET exposes this ability through the use of *attributes*.

You can modify a variety of attributes in the HTML of an element. For this example, we'll attach a bit of JavaScript to a button so that a message box pops up when the user clicks the button. First we need to create a page with a button control on it (Listing 9-27).

Listing 9-27. *Set HTML Attributes at Runtime*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Programmatic Controls</title>
</head>
<body>
    <form id="form1" runat="server">
    <asp:Button ID="btnSubmit" runat="server" Text="Click Me!" />
    </form>
</body>
</html>
```

If you run this code, you'll see output like the page in Figure 9-8. Clicking the button will simply cause the page to PostBack, but you won't see anything specific happen other than a quick flash.

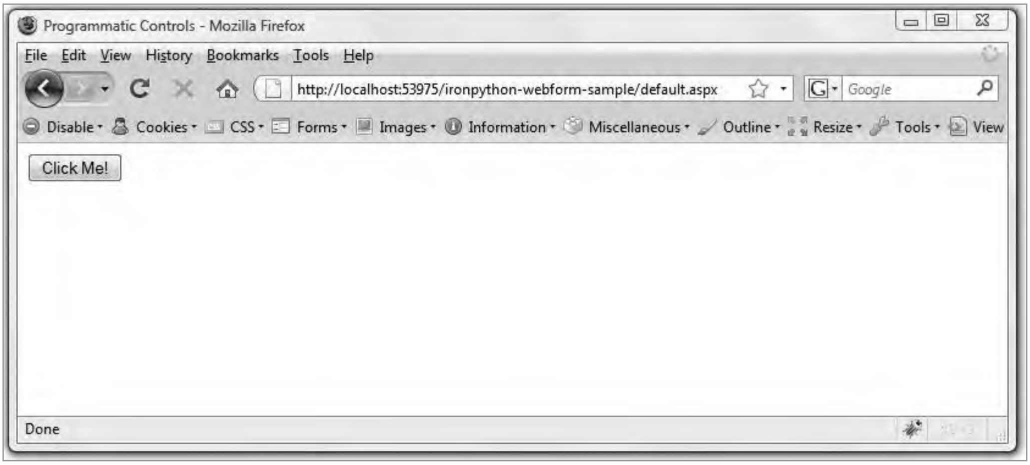


Figure 9-8. *Aside from PostBack, the button has little functionality.*

Now, in the code-behind file, we can modify the control to add a bit of JavaScript so that a message pops up when the button is clicked (Listing 9-28).

Listing 9-28. *Set HTML Attributes at Runtime (cont.)*

```
import System

def Page_Load(sender, e):
    btnSubmit.Attributes.Add("onclick", "javascript:alert('Added➥
dynamically!');return false;")
```

Run the code again, and when the button is clicked, the message "Added dynamically!" will appear for the user to click (Figure 9-9).

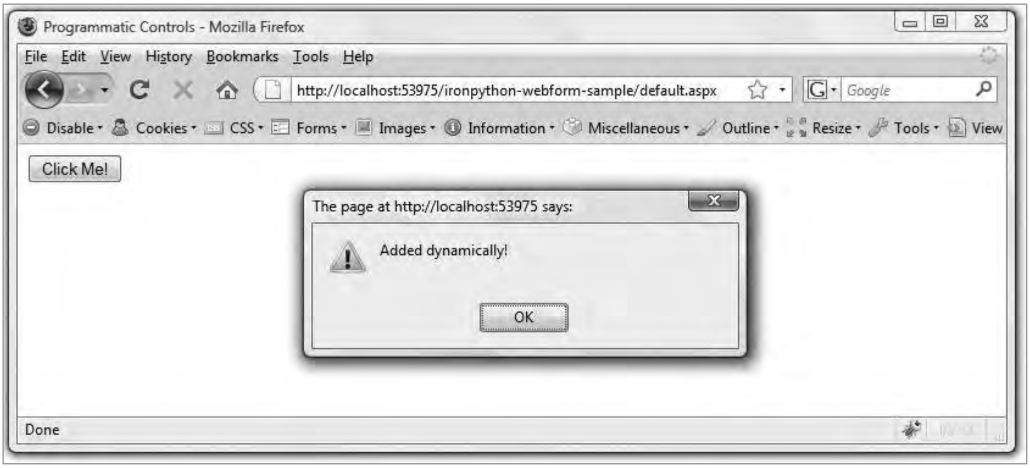


Figure 9-9. *Attaching JavaScript enhances the client-side behavior of the button.*

■ **Tip** Even if JavaScript seems like Latin to you, the function of the *alert()* method is probably pretty straightforward. But what's the deal with `return false;` at the end? If you remove this bit, you'll notice, the page will PostBack. Feel free to try it; we'll be using this functionality in the next snippet.

Using JavaScript to Determine Server-Side Operations

JavaScript isn't just for sending data to the client; we can also use it to get information back. One of the most straightforward methods is via the JavaScript *confirm()* method, which presents an OK and Cancel window for the user to make a decision.

We can use the same HTML markup from the previous example, but we'll modify the code-behind file to adjust the behavior of the button (Listing 9-29, Figure 9-10).

Listing 9-29. *Set HTML Attributes at Runtime*

```
import System

def Page_Load(sender, e):
    btnSubmit.Attributes.Add("onclick", "javascript:return confirm('Do you want➥
to confirm this operation?');")
```

```
if (IsPostBack):
    Response.Write("You confirmed the operation!")
else:
   pass
```

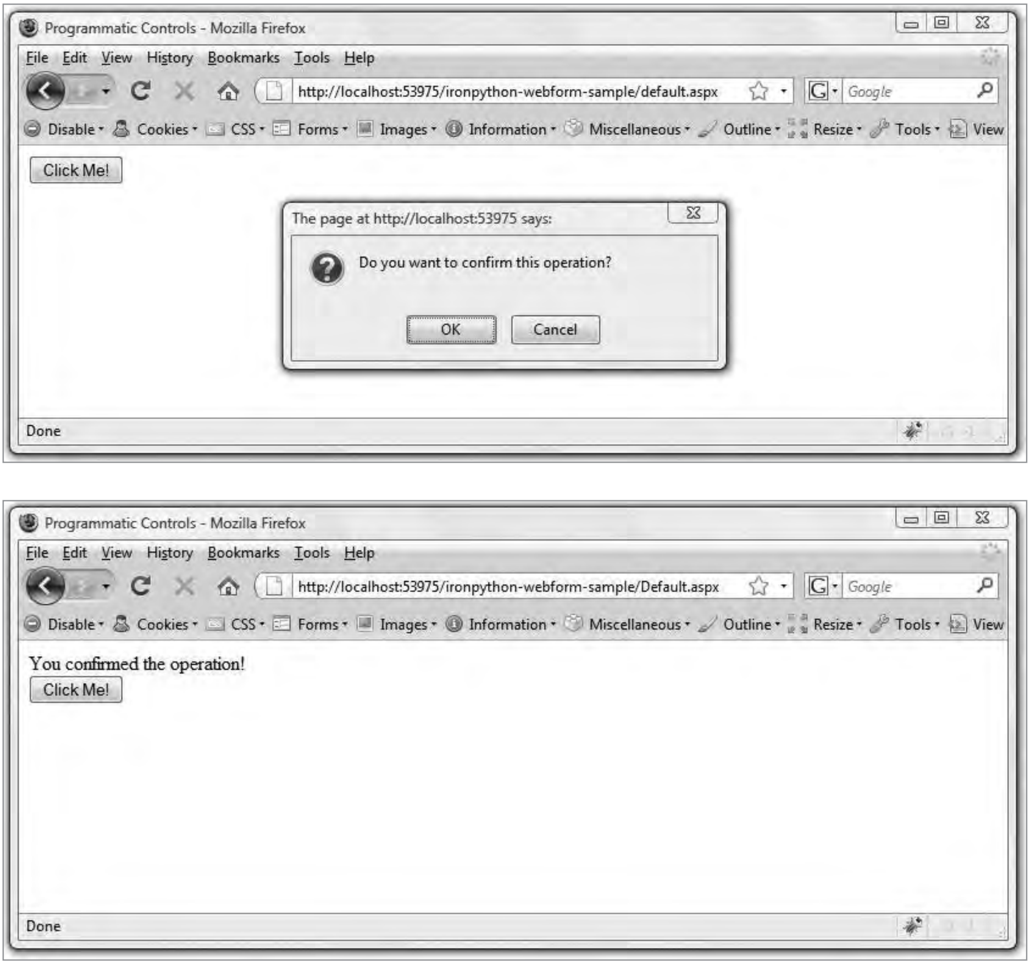


Figure 9-10. *The results of the confirmation are reflected in the code-behind operation.*

Screen Scraping

Screen scraping is the process of writing code to read markup from a web page by loading that page itself. It has a variety of uses, some legitimate, some not. One of the most well-known (and incredibly beneficial) uses is in search engines. Search engines read the

content of an HTML page, employ a variety of filtering techniques to extract keywords, and make logical decisions about how content is related.

Screen scraping in .NET is very simple. You only need to open a connection to the page, get the markup, and perform whatever processing you need to do. Let's open Google's home page and paste the content into a control on our page. First, we need to set up a page to hold that content (Listing 9-30).

Listing 9-30. *Screen Scrape*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Programmatic Controls</title>
</head>
<body>
    <form id="form1" runat="server">
    <asp:Literal ID="litContent" runat="server" />
    </form>
</body>
</html>
```

Now we can open Google's home page and render it on our page (Listing 9-31, Figure 9-11).

Listing 9-31. *Screen Scrape (cont.)*

```
import System
import System.Net
import System.IO

def Page_Load(sender, e):
    request = System.Net.HttpWebRequest.Create("http://www.google.com")
    response = request.GetResponse()
    sr = System.IO.StreamReader(response.GetResponseStream())
    litContent.Text = sr.ReadToEnd()
    sr.Close()
```



Figure 9-11. *Google's home page content scraped and displayed in our page*

Caution Why are the images missing? Google's home page is looking for them in the correct location relative to their document, which means the links won't work on our local machine. Screen scraping is an imperfect art! In general, it's not likely you'd be scraping in this manner; more often, it's to find links or content within a page and harvest that information for use elsewhere.

Setting the Default Button on a Form

One frequent task during application development is setting the default button on a form so that when the user presses the Enter key, a specific button event is fired. You can easily accomplish this by setting the *DefaultButton* attribute of the form on your page (Listing 9-32).

Listing 9-32. *Set the Default Button on a Form*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Programmatic Controls</title>
</head>
```

```
<body>
    <form id="form1" runat="server" defaultbutton="btnLogin">
         Username: <asp:TextBox ID="txtName" runat="server"></asp:TextBox>
         Password: <asp:TextBox ID="txtPassword" runat="server"➥
TextMode="Password"></asp:TextBox>
         <asp:Button ID="btnLogin" runat="server" Text="Login" />
         <asp:Button ID="btnCancel" runat="server" Text="Cancel" />
    </form>
</body>
</html>
```

■**Note** You don't need anything fancy in the *code-behind* file for this example; "pass" in the *Form_Load* method will work just fine.

Viewing Tracing Information About Pages

When debugging .NET applications, it can be very useful to view the trace information about your requests. This property can be set at the very top of an *.aspx* page (Listing 9-33, Figure 9-12).

Listing 9-33. *View Tracing Information About Pages*

```
<%@ Page Language="IronPython" CodeFile="Default.aspx.py" Trace="true" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Programmatic Controls</title>
</head>
<body>
    <form id="form1" runat="server">
    </form>
</body>
</html>
```

Request Details

Session Id:	tsvpnz55teyr0j55egsn0355	**Request Type:**	GET
Time of Request:	4/1/2009 12:01:51 PM	**Status Code:**	200
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin PreInit		
aspx.page	End PreInit	4.80507997524825E-05	0.000048
aspx.page	Begin Init	8.27619152713543E-05	0.000035
aspx.page	End Init	0.00127544143180209	0.001193
aspx.page	Begin InitComplete	0.00131168905545258	0.000036
aspx.page	End InitComplete	0.0013355747727714	0.000024
aspx.page	Begin PreLoad	0.00135757477556505	0.000022
aspx.page	End PreLoad	0.00137943509580128	0.000022
aspx.page	Begin Load	0.00140150493987364	0.000022
aspx.page	End Load	0.00145835574074359	0.000057
aspx.page	Begin LoadComplete	0.00148936526849083	0.000031
aspx.page	End LoadComplete	0.00151199384279287	0.000023
aspx.page	Begin PreRender	0.00153364463919297	0.000022
aspx.page	End PreRender	0.00155822876929889	0.000025
aspx.page	Begin PreRenderComplete	0.00158106686743706	0.000023
aspx.page	End PreRenderComplete	0.00160285734639458	0.000022
aspx.page	Begin SaveState	0.00183975896377892	0.000237
aspx.page	End SaveState	0.00195667326433946	0.000117
aspx.page	Begin SaveStateComplete	0.00198467961710217	0.000028
aspx.page	End SaveStateComplete	0.00201052089022487	0.000026
aspx.page	Begin Render	0.00203266057557595	0.000022
aspx.page	End Render	0.00226173996974476	0.000229

Control Tree

Control UniqueID	Type	Render Size Bytes (including children)	ViewState Size Bytes (excluding children)	ControlState Size Bytes (excluding children)

Figure 9-12. *Tracing information for recent requests*

Tip Although tracing can prove to be very useful in performance tuning and debugging, it's not something to use indiscriminately. Having this information assembled on every request brings with it a performance overhead cost, and I would advise staying away from its use in production applications.

Performing SEO-Friendly 301 Redirects

By default, the *Response.Redirect()* method in .NET is a temporary redirection from one resource to another. Used improperly, this can have a negative effect on the ranking that search engines assign to your pages; in general, a higher score is assigned to pages with many inbound links that have a fairly long history of being located at one spot. Many temporary redirections to a resource can indicate to the search engine that the content won't live there long, and your ranking suffers for it. A better choice is to opt for 301 redirects, which indicate that a resource has moved permanently (Listing 9-34).

Note This assumes of course that the resource *should* in fact have a permanent redirection; if it is content located temporarily at one location or is not to be indexed by search engines, then a temporary redirect won't do any harm. This is again a judgment call based on the particular situation.

Listing 9-34. *Perform SEO-friendly 301 Redirects*

```
newPath = "http://www.mydomain.com/someNewPage.aspx"

Response.Status = "301 Moved Permanently"
Response.AddHeader("Location", newPath)
```

Looping Through the Server Variables

A lot of information is hidden away in requests to and from the server. Some of this information is specific to your machine, some specific to the server itself, and a lot dedicated to the specific request at hand.

The *ServerVariables* collection in .NET stores all of this information in a format that makes retrieval straightforward and quick. For our example, we'll just loop through all the server variables and display their names to the screen (Listing 9-35, Figure 9-13).

Listing 9-35. *Loop Through the Server Variables*

```
import System

def Page_Load(sender, e):
    for i in Request.ServerVariables:
          Response.Write(i + "<br/>")
```

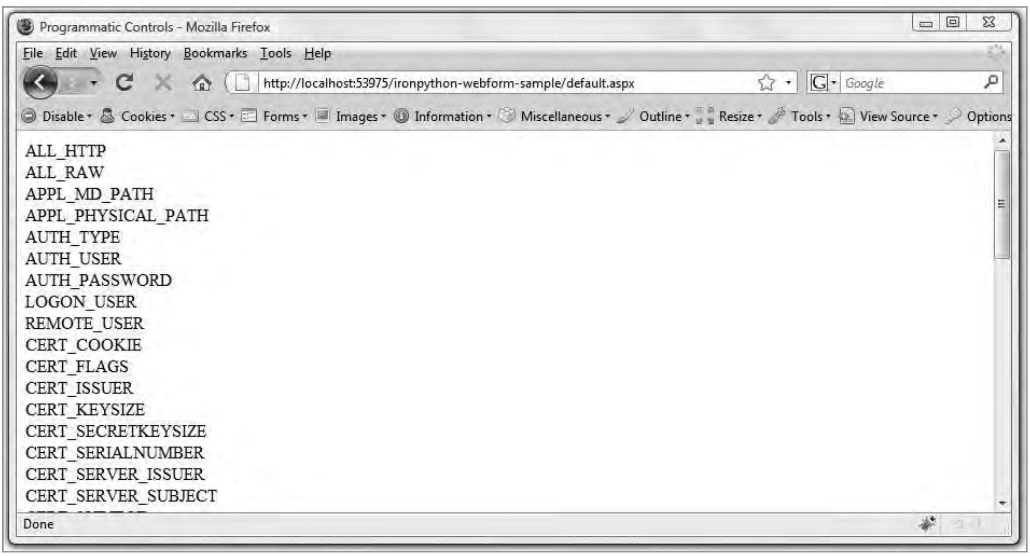


Figure 9-13. *The full list of server variables for this system*

■ **Tip** There are quite a few server variables, and some are unique to the server type, the hosting service (e.g., IIS, Apache), and the code running on that service. If you're wondering when you might ever use them, a good example might be logging who is visiting your web pages. You can use *Request.ServerVariables ["HTTP_REFERER"]* to find out the referring page, letting you know if users are coming from a search engine, a competitor's site, etc.

You may have noticed that the *_REFERER* part is spelled wrong. It should be *_REFERRER*, but this is such a common misspelling that it actually made it into the HTTP specification ages ago. You'll see it used quite frequently, because the developers behind browsers are trying harder and harder to stick to the HTTP specification documents.

Summary

That's it! We are officially done. Hopefully in this chapter I was able to provide you with a good number of reusable IronPython building blocks that you can easily drop into your applications and use as is or as starting points for more complex operations and designs. We covered console, desktop, and web application development tasks and specific tips and tricks within each type of development. I'd sincerely like to thank you for taking this journey down the dynamic language rabbit hole with me; IronPython is an exciting addition to .NET and one that is growing by leaps and bounds. For set-in-their-ways static-language developers like me, IronPython has been a real learning experience and a personal challenge to set the old ways aside for a bit and try something new. I hope it opens as many doors for you as it has for me.

Index

D

E

F

X

Y

Z